FAULT TOLERANT DESIGN: AN INTRODUCTION

ELENA DUBROVA
Department of Microelectronics and Information Technology
Royal Institute of Technology
Stockholm, Sweden

Kluwer Academic Publishers
Boston/Dordrecht/London

Contents

Acknowledgments

I would like to thank KTH students Jonian Grazhdani, Xavier Lowagie, Pieter Nuyts, Henrik Kirkeby, Chen Fu, Kareem Refaat, Sergej Koziner, Julia Kuznetsova, and Dr. Roman Morawek from Technikum Wien for reading and correcting the draft of the manuscript.

I am grateful to the Swedish Foundation for International Cooperation in Research and Higher Education (STINT) for the scholarship KU2002-4044 which supported my trip to the University of New South Wales, Sydney, Australia, where the first draft of this book was written during October - December 2002.

Chapter 1

INTRODUCTION

If anything can go wrong, it will.

—Murphy's law

1. Definition of fault tolerance

Fault tolerance is the ability of a system to continue performing its intended function in spite of faults. In a broad sense, fault tolerance is associated with reliability, with successful operation, and with the absence of breakdowns. A fault-tolerant system should be able to handle faults in individual hardware or software components, power failures or other kinds of unexpected disasters and still meet its specification.

Fault tolerance is needed because it is practically impossible to build a perfect system. The fundamental problem is that, as the complexity of a system increases, its reliability drastically deteriorates, unless compensatory measures are taken. For example, if the reliability of individual components is 99.99%, then the reliability of a system consisting of 100 non-redundant components is 99.01%, whereas the reliability of a system consisting of 10.000 non-redundant components is just 36.79%. Such a low reliability is unacceptable in most applications. If a 99% reliability is reqiured for a 10.000 component system, the individual components with the reliability of at least 99.999% should be used, implying the increase in cost.

Another problem is that, although designers do their best to have all the hardware defects and software bugs cleaned out of the system before it goes on the market, history shows that such a goal is not attainable. It is inevitable that some unexpected environmental factor is not taken into account, or some potential user mistakes are not foreseen. Thus, even in the unlikely case that a

system is designed and implemented perfectly, faults are likely to be caused by situations out of the control of the designers.

A system is said to fail if it ceased to perform its intended function. *System* is used in this book in a generic sense of a group of independent but interrelated elements comprising a unified whole. Therefore, the techniques presented are also applicable to the variety of products, devices and subsystems. *Failure* can be a total cessation of function, or a performance of some function in a subnormal quality or quantity, like deterioration or instability of operation. The aim of fault-tolerant design is to minimize the probability of failures, whether those failures simply annoy the customers or result in lost fortunes, human injury or environmental disaster.

exist to increase component reliability. Failure rates in hardware are

2. Fault tolerance and redundancy

There are various approaches to achieve fault-tolerance. Common to all these approaches is a certain amount of redundancy. For our purposes, *redundancy* is the provision of functional capabilities that would be unnecessary in a fault-free environment. This can be a replicated hardware component, an additional check bit attached to a string of digital data, or a few lines of program code verifying the correctness of the program's results. The idea of incorporating redundancy in order to improve reliability of a system was pioneered by John von Neumann in early 1950s in his work "Probabilistic logic and the synthesis of reliable organisms from unreliable components".

Two kinds of redundancy are possible: space redundancy and time redundancy. *Space redundancy* provides additional components, functions, or data items that are unnecessary for a fault-free operation. Space redundancy is further classified into hardware, software and information redundancy, depending on the type of redundant resources added to the system. In *time redundancy* the computation or data transmission is repeated and the result is compared to a stored copy of the previous result.

3. Applications of fault-tolerance

Originally, fault-tolerance techniques were used to cope with physical defects of individual hardware components. Designers of early computing systems employed redundant structures with voting to eliminate the effect of failed components, error-detection or correcting codes to detect or correct information errors, diagnostic techniques to locate failed components and automatic switch-overs to replace them.

Following the development of semiconductor technology, hardware components became intrinsically more reliable and the need for tolerance of component defect diminished in general purpose applications. Nevertheless, fault tolerance

remained necessary in many safety-, mission- and business-critical applications. *Safety-critical* applications are those where loss of life or environmental disaster must be avoided. Examples are nuclear power plant control systems, computer-controlled radiation therapy machines or heart pace-makers, military radar systems. *Mission-critical* applications stress mission completion, as in case of an airplane or a spacecraft. *Business-critical* are those in which keeping a business operating is an issue. Examples are bank and stock exchange's automated trading system, web servers, e-commerce.

As complexity of systems grew, a need to tolerate other than hardware component faults has aroused. The rapid development of real-time computing applications that started around the mid-1990s, especially the demand for software-embedded intelligent devices, made software fault tolerance a pressing issue. Software systems offer compact design, rich functionality and competitive cost. Instead of implementing a given functionality in hardware, the design is done by writing a set of instructions accomplishing the desired tasks and loading them into a processor. If changes in the functionality are needed, the instructions can be modified instead of building a different physical device.

An inevitable related problem is that the design of a system is performed by someone who is not an expert in that system. For example, the autopilot expert decides how the device should work, and then provides the information to a software engineer, who implements the design. This extra communication step is the source of many faults in software today. The software is doing what the software engineer thought it should do, rather than what the original design engineer required. Nearly all the serious accidents in which software has been involved in the past can be traced to this origin.

Chapter 2

FUNDAMENTALS OF DEPENDABILITY

Ah, this is obviously some strange usage of the word 'safe' that I wasn't previously aware of.

—Douglas Adams, "The Hitchhikers Guide to the Galaxy".

1. Introduction

The ultimate goal of fault tolerance is the development of a dependable system. In a broad term, *dependability* is the ability of a system to deliver its intended level of service to its users. As computer systems become relied upon by society more and more, the dependability of these systems becomes a critical issue. In airplanes, chemical plants, heart pace-makers or other safety critical applications, a system failure can cost people's lives or environmental disaster.

In this section, we study three fundamental characteristics of dependability: attributes, impairment and means. Dependability *attributes* describe the properties which are required from a system. Dependability *impairments* express the reasons for a system to cease to perform its function or, in other words, the threats to dependability. Dependability *means* are the methods and techniques enabling the development of a dependable computing system.

2. Dependability attributes

The attributes of dependability express the properties which are expected from a system. Three primary attributes are reliability, availability and safety. Other possible attributes include maintainability, testability, performability, confidentiality, security. Depending on the application, one or more of these attributes are needed to appropriately evaluate the system behavior. For example, in an automatic teller machine (ATM), the proportion of time which system is able to deliver its intended level of service (system availability) is an important

measure. For a cardiac patient with a pacemaker, continuous functioning of the device is a matter of life and death. Thus, the ability of the system to deliver its service without interruption (system reliability) is crucial. In a nuclear power plant control system, the ability of the system to perform its functions correctly or to discontinue its function in a safe manner (system safety) is of greater importance.

2.1 Reliability

Reliability $R(t)$ of a system at time t is the probability that the system operates without failure in the interval $[0,t]$, given that the system was performing correctly at time 0.

Reliability is a measure of the continuous delivery of correct service. High reliability is required in situations when a system is expected to operate without interruptions, as in the case of a pacemaker, or when maintenance cannot be performed because the system cannot be accessed. For example, spacecraft mission control system is expected to provide uninterrupted service. A flaw in the system is likely to cause a destruction of the spacecraft as in the case of NASA's earth-orbiting Lewis spacecraft launched on August 23rd, 1997. The spacecraft entered a flat spin in orbit that resulted in a loss of solar power and a fatal battery discharge. Contact with the spacecraft was lost, and it then re-entered the atmosphere and was destroyed on September 28th. According to the report of the Lewis Spacecraft Mission Failure Investigation, the failure was due to a combination of a technically flawed attitude-control system design and inadequate monitoring of the spacecraft during its crucial early operations phase.

Reliability is a function of time. The way in which time is specified varies considerably depending on the nature of the system under consideration. For example, if a system is expected to complete its mission in a certain period of time, like in case of a spacecraft, time is likely to be defined as a calendar time or as a number of hours. For software, the time interval is often specified in so called *natural or time units*. A natural unit is a unit related to the amount of processing performed by a software-based product, such as pages of output, transactions, telephone calls, jobs or queries.

2.2 Availability

Relatively few systems are designed to operate continuously without interruption and without maintenance of any kind. In many cases, we are interested not only in the probability of failure, but also in the number of failures and, in particular, in the time required to make repairs. For such applications, the attribute which we would like to maximize is the fraction of time that the system is in the operational state, expressed by availability.

Availability $A(t)$ of a system at time t is the probability that the system is functioning correctly at the instant of time t.

$A(t)$ is also referred as *point* availability, or *instantaneous* availability. Often it is necessary to determine the *interval* or *mission* availability. It is defined by

$$A(T) = \frac{1}{T}\int_0^T A(t)dt. \tag{2.1}$$

$A^(T)$ is the value of the point availability averaged over some interval of time T. This interval might be the life-time of a system or the time to accomplish some particular task. Finally, it is often found that after some initial transient effect, the point availability assumes a time-independent value. In this case, the *steady-state* availability is defined by

$$A(\infty) = \lim_{T\to\infty}\frac{1}{T}\int_0^T A(t)dt. \tag{2.2}$$

If a system cannot be repaired, the point availability $A(t)$ is equal to the system's reliability, i.e. the probability that the system has not failed between 0 and t. Thus, as T goes to infinity, the steady-state availability of a non-repairable system goes to zero

$$A(\infty) = 0$$

Steady-state availability is often specified in terms of *downtime per year*. Table 2.1 shows the values for the availability and the corresponding downtime.

Availability	Downtime
90%	36.5 days/year
99%	3.65 days/year
99.9%	8.76 hours/year
99.99%	52 minutes/year
99.999%	5 minutes/year
99.9999%	31 seconds/year

Table 2.1. Availability and the corresponding downtime per year.

Availability is typically used as a measure for systems where short interruptions can be tolerated. Networked systems, such as telephone switching and web servers, fall into this category. A customer of a telephone system expects to complete a call without interruptions. However, a downtown of three minutes a year is considered acceptable. Surveys show that web users lose patience when web sites take longer than eight seconds to show results. This means

that such web sites should be available all the time and should respond quickly even when a large number of clients concurrently access them. Another example is the electrical power control system. Customers expect power to be available 24 hours a day, every day, in any weather condition. In some cases, a prolonged power failure may lead to health hazards, due to the loss of services such as water pumps, heating, light, or medical attention. Industries may suffer substantial financial loss.

2.3 Safety

Safety can be considered as an extension of reliability, namely a reliability with respect to failures that may create safety hazards. From the reliability point of view, all failures are equal. On the other hand, for safety considerations, failures are partitioned into *fail-safe* and *fail-unsafe* ones.

As an example consider an alarm system. The alarm may either fail to function even though a dangerous situation exists, or it may give a false alarm when no danger is present. The former is classified as a fail-unsafe failure. The latter is considered a fail-safe one. More formally, safety is defined as follows.

Safety $S(t)$ of a system is the probability that the system will either perform its function correctly or will discontinue its operation in a fail-safe manner.

Safety is required in *safety-critical applications* were a failure may result in an human injury, loss of life or environmental disaster. Examples are chemical or nuclear power plant control systems, aerospace and military applications.

Many unsafe failures are caused by human mistakes. For example, the Chernobyl accident on April 26th, 1986, happened because all safety systems were shut off to allow an experiment which aimed investigating a possibility of producing electricity from the residual energy in the turbo-generators. The experiment was badly planned, and was led by an electrical engineer who was not familiar with the reactor facility. The experiment could not be canceled when things went wrong, because all automatic shutdown systems and the emergency core cooling system of the reactor had been manually turned off.

3. Dependability impairments

Dependability impairment are usually defined in terms of faults, errors, failures. A common feature of the three terms is that they give us a message that something went wrong. A difference is that, in case of a fault, the problem occurred on the physical level; in case of an error, the problem occurred on the computational level; in case of a failure, the problem occurred on a system level.

3.1 Faults, errors and failures

A *fault* is a physical defect, imperfection, or flaw that occurs in some hardware or software component. Examples are short-circuit between two adjacent interconnects, broken pin, or a software bug.

An *error* is a deviation from correctness or accuracy in computation, which occurs as a result of a fault. Errors are usually associated with incorrect values in the system state. For example, a circuit or a program computed an incorrect value, an incorrect information was received while transmitting data.

A *failure* is a non-performance of some action which is due or expected. A system is said to have a failure if the service it delivers to the user deviates from compliance with the system specification for a specified period of time. A system may fail either because it does not act in accordance with the specification, or because the specification did not adequately describe its function.

Faults are reasons for errors and errors are reasons for failures. For example, consider a power plant, in which a computer controlled system is responsible for monitoring various plant temperatures, pressures, and other physical characteristics. The sensor reporting the speed at which the main turbine is spinning breaks. This fault causes the system to send more steam to the turbine than is required (error), over-speeding the turbine, and resulting in the mechanical safety system shutting down the turbine to prevent damaging it. The system is no longer generating power (system failure, fail-safe).

The definitions of physical, computational and system level are a bit more confusing when applied to software. In the context of this book, we interpret a program code as physical level, the values of a program state as computational level, and the software system running the program as system level. For example, an operating system is a software system. Then, a bug in a program is a fault, possible incorrect value caused by this bug is an error and possible crush of the operating system is a failure.

Not every fault causes an error and not every error causes a failure. This is particularly evident in the software case. Some program bugs are very hard to find because they cause failures only in very specific situations. For example, in November 1985, $32 billion overdraft was experienced by the Bank of New York, leading to a loss of $5 million in interests. The failure was caused by an unchecked overflow of an 16-bit counter. In 1994, Intel Pentium I microprocessor was discovered to compute incorrect answers to certain floating-point division calculations. For example, dividing 5505001 by 294911 produced 18.66600093 instead of 18.66665197. The problem had occurred because of the omission of five entries in a table of 1066 values used by the division algorithm. The five cells should have contained the constant +2, but because the cells were empty, the processor treated them as a zero.

3.2 Origins of faults

As we discussed earlier, failures are caused by errors and errors are caused by faults. Faults are, in turn, caused by numerous problems occurring at specification, implementation, fabrication stages of the design process. They can also be caused by external factors, such as environmental disturbances or human actions, either accidental or deliberate. Broadly, we can classify the sources of faults into four groups: incorrect specification, incorrect implementation, fabrication defects and external factors.

Incorrect specification results from incorrect algorithms, architectures, or requirements. A typical example is a case when the specification requirements ignore aspects of the environment in which the system operates. The system might function correctly most of the time, but there also could be instances of incorrect performance. Faults caused by incorrect specifications are usually called *specification faults*. In System-on-a-Chip design, integrating pre-designed intellectual property (IP) cores, specification faults are one of the most common type of faults. Core specifications, provided by the core vendors, do not always contain all the details that system-on-a-chip designers need. This is partly due to the intellectual property protection requirements, especially for core netlists and layouts.

Faults due to *incorrect implementation*, usually referred to as *design faults*, occur when the system implementation does not adequately implement the specification. In hardware, these include poor component selection, logical mistakes, poor timing or synchronization. In software, examples of incorrect implementation are bugs in the program code and poor software component reuse. Software heavily relies on different assumptions about its operating environment. Faults are likely to occur if these assumptions are incorrect in the new environment. The Ariane 5 rocket accident is an example of a failure caused by a reused software component. Ariane 5 rocket exploded 37 seconds after lift-off on June 4th, 1996, because of a software fault that resulted from converting a 64-bit floating point number to a 16-bit integer. The value of the floating point number happened to be larger than the one that can be represented by a 16-bit integer. In response to the overflow, the computer cleared its memory. The memory dump was interpreted by the rocket as an instruction to its rocket nozzles, which caused an explosion.

A source of faults in hardware are *component defects*. These include manufacturing imperfections, random device defects and components wear-outs. Fabrication defects were the primary reason for applying fault-tolerance techniques to early computing systems, due to the low reliability of components. Following the development of semiconductor technology, hardware components became intrinsically more reliable and the percentage of faults caused by fabrication defects diminished.

The fourth cause of faults are *external factors*, which arise from outside the system boundary, the environment, the user or the operator. External factors include phenomena that directly affect the operation of the system, such as temperature, vibration, electrostatic discharge, nuclear or electromagnetic radiation or that affect the inputs provided to the system. For instance, radiation causing a bit to flip in a memory location is a fault caused by an external factor. Faults caused by user or operator mistakes can be accidental or malicious. For example, a user can accidentally provide incorrect commands to a system that can lead to system failure, e.g. improperly initialized variables in software. Malicious faults are the ones caused, for example, by software viruses and hacker intrusions.

3.3 Common-mode faults

A *common-mode fault* is a fault which occurs simultaneously in two or more redundant components. Common-mode faults are caused by phenomena that create dependencies between the redundant units which cause them to fail simultaneously, i.e. common communication buses or shared environmental factors. Systems are vulnerable to common-mode faults if they rely on a single source of power, cooling or input/output (I/O) bus.

Another possible source of common-mode faults is a design fault which causes redundant copies of hardware or of the same software process to fail under identical conditions. The only fault-tolerance approach for combating common-mode design faults is design diversity. *Design diversity* is the implementation of more than one variant of the function to be performed. For computer-based applications, it is shown to be more efficient to vary a design at higher levels of abstractions. For example, varying algorithms is more efficient than varying implementation details of a design, e.g. using different program languages. Since diverse designs must implement a common system specification, the possibility for dependency always arises in the process of refining the specification. Truly diverse designs eliminate dependencies by using separate design teams, different design rules and software tools.

3.4 Hardware faults

In this section we first consider two major classes of hardware faults: permanent and transient faults. Then, we show how different types of hardware faults can be modeled.

3.4.1 Permanent and transient faults

Hardware faults are classified with respect to fault duration into permanent, transient and intermittent faults.

A *permanent fault* remains active until a corrective action is taken. These faults are usually caused by some physical defects in the hardware, such as shorts in a circuit, broken interconnections or stuck bits in the memory. Permanent faults can be detected by on-line test routines that work concurrently with the normal system operation.

A *transient fault* remains active for a short period of time. A transient fault that becomes active periodically is an *intermittent fault*. Because of their short duration, transient faults are often detected through the errors that result from their propagation. Transient faults are often called *soft faults* or *glitches*. Transient fault are dominant type of faults in computer memories. For example, about 98% of RAM faults are transient faults. The causes of transient faults are mostly environmental, such as alpha particles, cosmic rays, electrostatic discharge, electrical power drops, overheating or mechanical shock. For instance, a voltage spike might cause a sensor to report an incorrect value for a few milliseconds before reporting correctly. Studies show that a typical computer experiences more than 120 power problems per month. Cosmic rays cause the failure rate of electronics at airplane altitudes to be approximately one hundred times greater than at sea level. Intermittent faults can be due to implementation flaws, aging and wear-out, and to unexpected operation conditions. For example, a loose solder joint in combination with vibration can cause an intermittent fault.

3.4.2 Fault models

It is not possible to enumerate all possible types of faults which can occur in a system. To make the evaluation of fault coverage possible, faults are assumed to behave according to some *fault model*. Some of the commonly used fault models are: stuck-at fault, transition fault, coupling fault. A fault model attempts to describe the effect of the fault that can occur.

A *stuck-at fault* is a fault which results in a line in the circuit or a memory cell being permanently stuck at a logic one or zero. It is assumed that the basic functionality of the circuit is not changed by the fault, i.e. a combinational circuit is not transformed to a sequential circuit, or an AND gate does not become an OR gate. Due to its simplicity and effectiveness, stuck at fault is the most common fault model.

A *transition fault* is a fault in which line in the circuit or a memory cell cannot change from a particular state to another state. For example, suppose a memory cell contains a value zero. If a one is written to the cell, the cell successfully changes its state. However, a subsequent write of a zero to the cell does not change the state of the cell. The memory is said to have a one-to-zero transition fault. Both stuck-at faults and transition faults can be easily detected during testing.

Coupling faults are more difficult to test because they depend upon more than one line. An example of a coupling fault would be a short-circuit between two adjacent word lines in a memory. Writing a value to a memory cell connected to one of the word lines would also result in that value being written to the corresponding memory cell connected to the other short-circuited word line. Two types of transition coupling faults include *inversion* coupling faults in which a specific transition in one memory cell inverts the contents of another memory cell, and *idempotent* coupling faults in which a specific transition of one memory cell results in a particular value (0 or 1) being written to another memory cell.

Clearly, fault models are not accurate in 100% cases, becuase faults can cause a variety of different effects. However, studies have shown that a combination of several fault models can give a very precise coverage of actual faults. For example, for memories, practically all faults can be modeled as a combination of stuck-at faults, transition faults and idempotent coupling faults.

3.5 Software faults

Software differs from hardware in several aspects. First, software does not age or wear out. Unlike mechanical or electronic parts of hardware, software cannot be deformed, broken or affected by environmental factors. Assuming that software is deterministic, it will always behave the same way in the same circumstances, unless there are problems in hardware that change the storage content or data path. Since the software does not change once it is uploaded into memory and starts running, trying to achieve fault tolerance by simply replicating the same software modules will not work, because all copies will have identical faults.

Second, software may undergo several upgrades during the system life cycle. These can be either reliability upgrades or feature upgrades. A *reliability upgrade* targets to enhance software reliability or security. This is usually done by re-designing or re-implementing some modules using better engineering approaches. A *feature upgrade* aims to enhance the functionality of the software. It is likely to increase the complexity and thus decrease the reliability by possibly introducing additional faults into the software.

Third, fixing bugs does not necessarily make the software more reliable. On the contrary, new unexpected problems may arise. For example, in 1991, a change of three lines of code in a signaling program containing millions of lines of code caused the local telephone systems in California and along the Eastern coast to stop.

Finally, since software is inherently more complex and less regular than hardware, achieving sufficient verification coverage is more difficult. Traditional testing and debugging methods are inadequate for large software systems. The recent focus on formal methods promises higher coverage, however, due to their

extremely large computational complexity they are only applicable in specific applications. Due to incomplete verification, most of the software faults are design faults, occurring when a programmer either misunderstands the specification or simply makes a mistake. Design faults are related to fuzzy human factors, and therefore they are harder to prevent. In hardware, design faults may also exist, but other types of faults, such as fabrication defects and transient faults caused by environmental factors, usually dominate.

4. Dependability means

Dependability means are the methods and techniques enabling the development of a dependable system. Fault tolerance, which is the subject of this book, is one of such methods. It is normally used in a combination with other methods to attain dependability, such as fault prevention, fault removal and fault forecasting. Fault prevention aims to prevent the occurrences or introduction of faults. Fault removal aims to reduce the number of faults which are present in the system. Fault forecasting aims to estimate how many faults are present, possible future occurrences of faults, and the impact of the faults on the system.

4.1 Fault tolerance

Fault tolerance targets the development of systems which function correctly in presence of faults. Fault tolerance is achieved by using some kind of redundancy. In the context of this book, *redundancy* is the provision of functional capabilities that would be unnecessary in a fault-free environment. The redundancy allows either to *mask* a fault, or to *detect* a fault, with the following location, containment and recovery.

Fault masking is the process of insuring that only correct values get passed to the system output in spite of the presence of a fault. This is done by preventing the system from being affected by errors by either correcting the error, or compensating for it in some fashion. Since the system does not show the impact of the fault, the existence of fault is therefore invisible to the user/operator. For example, a memory protected by an error-correcting code corrects the faulty bits before the system uses the data. Another example of fault masking is triple modular redundancy with majority voting.

Fault detection is the process of determining that a fault has occurred within a system. Examples of techniques for fault detection are acceptance tests and comparison. *Acceptance tests* are common in processors. The result of a program is subjected to a test. If the result passes the test, the program continues execution. A failed acceptance test implies a fault. *Comparison* is used for systems with duplicated components. A disagreement in the results indicates the presence of a fault.

Fault location is the process of determining where a fault has occurred. A failed acceptance test cannot generally be used to locate a fault. It can only tell that something has gone wrong. Similarly, when a disagreement occurs during the comparison of two modules, it is not possible to tell which of the two has failed.

Fault containment is the process of isolating a fault and preventing the propagation of the effect of that fault throughout the system. The purpose is to limit the spread of the effects of a fault from one area of the system into another area. This is typically achieved by frequent fault detection, by multiple request/confirmation protocols and by performing consistency checks between modules.

Once a faulty component has been identified, a system *recovers* by reconfiguring itself to isolate the component from the rest of the system and regain operational status. This might be accomplished by having the component replaced, by marking it off-line and using a redundant system. Alternately, the system could switch it off and continue operation with a degraded capability. This is known as *graceful degradation*.

4.2 Fault prevention

Fault prevention is achieved by quality control techniques during specification, implementation and fabrication stages of the design process. For hardware, this includes design reviews, component screening and testing. For software, this includes structural programming, modularization and formal verification techniques.

A rigorous design review may eliminate many of the specification faults. If a design is efficiently tested, many of its faults and component defects can be avoided. Faults introduced by external disturbances such as lightning or radiation are prevented by shielding, radiation hardening, etc. User and operation faults are avoided by training and regular procedures for maintenance. Deliberate malicious faults caused by viruses or hackers are reduced by firewalls or similar security means.

4.3 Fault removal

Fault removal is performed during the development phase as well as during the operational life of a system. During the development phase, fault removal consists of three steps: verification, diagnosis and correction. Fault removal during the operational life of the system consists of corrective and preventive maintenance.

Verification is the process of checking whether the system meets a set of given conditions. If it does not, the other two steps follow: the fault that prevents the

conditions from being fulfilled is diagnosed and the necessary corrections are performed.

In *preventive maintenance*, parts are replaced, or adjustments are made before failure occurs. The objective is to increase the dependability of the system over the long term by staving off the aging effects of wear-out. In contrast, *corrective maintenance* is performed after the failure has occurred in order to return the system to service as soon as possible.

4.4 Fault forecasting

Fault forecasting is done by performing an evaluation of the system behavior with respect to fault occurrences or activation. The evaluation can be *qualitative*, that aims to rank the failure modes or event combinations that lead to system failure, or *quantitative*, that aims to evaluate in terms of probabilities the extent to which some attributes of dependability are satisfied, or coverage. Informally, *coverage* is the probability of a system failure given that a fault occurs. Simplistic estimates of coverage merely measure redundancy by accounting for the number of redundant success paths in a system. More sophisticated estimates of coverage account for the fact that each fault potentially alters a system's ability to resist further faults. We study qualitative and quantitative evaluation techniques in more details in the next section.

5. Problems

2.1. What is the primary goal of fault tolerance?

2.2. Give three examples of applications in which a system failure can cost people's lives or environmental disaster.

2.3. What is dependability of a system? Why the dependability of computer systems is a critical issue nowadays?

2.4. Describe three fundamental characteristics of dependability.

2.5. What do the attributes of dependability express? Why different attributes are used in different applications?

2.6. Define the reliability of a system. What property of a system the reliability characterizes? In which situations is high reliability required?

2.7. Define point, interval and steady-state availabilities of a system. Which attribute we would like to maximize in applications requiring high availability?

2.8. What is the difference between the reliability and the availability of a system? How does the point availability compare to the system's reliability if the

system cannot be repaired? What is the steady-state availability of a non-repairable system?

2.9. Compute the downtime per year for $A(\infty) = 80\%, 75\%$ and 50%.

2.10. A telephone system has less than 3 min per year downtime. What is its steady-state availability?

2.11. Define the safety of a system. Into which two groups the failures are partitioned for safety analysis? Give example of applications requiring high safety.

2.12. What are dependability impairments?

2.13. Explain the differences between faults, errors and failures and the relationships between them.

2.14. Describe the four major groups of fault sources. Give an example for each group. In your opinion, which of the groups causes "most expensive" faults?

2.15. What is a common-mode fault? By what kind of phenomena common-mode faults are caused? Which systems are most vulnerable to common-mode faults? Give examples.

2.16. How are hardware faults classified with respect to fault duration? Give an example for each type of faults.

2.17. Why are fault models introduced? Can fault models guarantee 100% accuracy?

2.18. Give an example of a combinational logic circuit in which a single stuck-at fault on a given line never causes an error on the output.

2.19. Suppose that we modify the stuck-at fault model in the following way. Instead of having a line being permanently stuck at a logic one or zero value, we have a transistor being permanently open or closed. Draw a transistor-level circuit diagram of a CMOS NAND gate.

(a) Give an example of a fault in your circuit which can be modeled by the new model but cannot be modeled by the standard stuck-at fault model.

(b) Find a fault in your circuit which cannot be modeled by the new model but can be modeled by the standard stuck-at fault model.

2.20. Explain main differences between software and hardware faults.

2.21. What are dependability means? What are the primary goals of fault prevention, fault removal and fault forecasting?

2.22. What is redundancy? Is redundancy necessary for fault-tolerance? Will any redundant system be fault-tolerant?

2.23. Does a fault need to be detected to be masked?

2.24. Define fault containment. Explain why fault containment is important.

2.25. Define graceful degradation. Give example of application where graceful degradation is desirable.

2.26. How is fault prevention achieved? Give examples for hardware and for software.

2.27. During which phases of system's life is fault removal performed?

2.28. What types of faults are targeted by verification?

2.29. What are the objectives of preventive and corrective maintenances?

2.30. Consider the logic circuit shown on p. 108, Fig. 6.2 (full adder). Ignore the s-a-1 fault shown on the picture, i.e. the circuit you analyze does not have this fault.

(a) Find a test for stuck-at-1 fault on the input *b*.

(b) Find a test for stuck-at-0 fault on the fan-out branch of the input *a* which feeds into an AND gate (lower input of the AND gate whose output is marked "s-a-1" on the picture).

Chapter 3

DEPENDABILITY EVALUATION TECHNIQUES

A common mistake that people make when trying to design something completely foolproof is to underestimate the ingenuity of complete fools.

—Douglas Adams, Mostly Harmless

1. Introduction

Along with cost and performance, dependability is the third critical criterion based on which system-related decisions are made. Dependability evaluation is important because it helps identifying which aspect of the system behavior, e.g. component reliability, fault coverage or maintenance strategy plays a critical role in determining overall system dependability. Thus, it provides a proper focus for product improvement effort from early in the development stage to fabrication and test.

There are two conventional approaches to dependability evaluation: (1) modeling of a system in the design phase, or (2) assessment of the system in a later phase, typically by test. The first approach relies on probabilistic models that use component level failure rates published in handbooks or supplied by the manufacturers. This approach provides an early indication of system dependability, but the model as well as the underlying data later need to be validated by actual measurements. The second approach typically uses test data and reliability growth models. It involves fewer assumptions than the first, but it can be very costly. The higher the dependability required for a system, the longer the test. A further difficulty arises in the translation of reliability data obtained by test into those applicable to the operational environment.

Dependability evaluation has two aspects. The first is *qualitative evaluation*, that aims to identify, classify and rank the failure modes, or the event combinations that would lead to system failures. For example, component faults or

environmental conditions are analyzed. The second aspect is *quantitative evaluation*, that aims to evaluate in terms of probabilities the extent to which some attributes of dependability, such as reliability, availability, safety, are satisfied. Those attributes are then viewed as measures of dependability.

In this chapter we study common dependability measures, such as failure rate, mean time to failure, mean time to repair, etc. Examining the time dependence of failure rate and other measures allows us to gain additional insight into the nature of failures. Next, we examine possibilities for modeling of system behaviors using reliability block diagrams and Markov processes. Finally, we show how to use these models to evaluate system's reliability, availability and safety.

We begin with a brief introduction into the probability theory, necessary to understand the presented material.

2. Basics of probability theory

Probability is the branch of mathematics which studies the possible outcomes of given events together with their relative likelihoods and distributions. In common language, the word "probability" is used to mean the chance that a particular event will occur expressed on a linear scale from 0 (impossibility) to 1 (certainty).

The first axiom of probability theory states that the value of probability of an event A lies between 0 and 1:

$$0 \leq p(A) \leq 1. \tag{3.1}$$

Let $\overline{A}$ denotes the event "not A". For example, if A stands for "it rains", $\overline{A}$ stands for "it does not rain". The second axiom of probability theory says that the probability of an event $\overline{A}$ is equal to 1 minus the probability of the event A:

$$p(\overline{A}) = 1 - p(A). \tag{3.2}$$

Suppose that one event, A is dependent on another event, B. Then $P(A|B)$ denotes the conditional probability of event A, given event B. The fourth rule of probability theory states that the probability $p(A \cdot B)$ that both A and B will occur is equal to the probability that B occur times the conditional probability $P(A|B)$:

$$p(A \cdot B) = p(A|B) \cdot p(B), \text{ if } A \text{ depends on } B. \tag{3.3}$$

If $p(B)$ is greater than zero, the equation 3.3 can be written as

$$p(A|B) = \frac{p(A \cdot B)}{p(B)} \tag{3.4}$$

An important condition that we will often assume is that two events are mutually independent. For events A and B to be independent, the probability $p(A)$ does not depend on whether B has already occurred or not, and vice versa. Thus, $p(A|B) = p(A)$. So, for independent events, the rule (3.3) reduces to

$$p(A \cdot B) = p(A) \cdot p(B), \text{ if } A \text{ and } B \text{ are independent events.} \tag{3.5}$$

This is the definition of independence, that the probability of two events both occurring is the product of the probabilities of each event occurring. Situations also arise when the events are mutually exclusive. That is, if A occurs, B cannot, and vice versa. So,$p(A \cdot B) = 0$ and $p(B \cdot A) = 0$ and the equation 3.3 becomes

$$p(A \cdot B) = 0, \text{ if } A \text{ and } B \text{ are mutually exclusive events.} \tag{3.6}$$

This is the definition of mutually exclusiveness, that the probability of two events both occurring is zero.

Let us now consider the situation when either A, or B, or both event may occur. The probability $p(A+B)$ is given by

$$p(A+B) = p(A) + p(B) - p(A \cdot B) \tag{3.7}$$

Combining (3.6) and (3.7), we get

$$p(A+B) = p(A) + p(B), \text{ if } A \text{ and } B \text{ are mutually exclusive events.} \tag{3.8}$$

As an example, consider a system consisting of three identical components A, B and C, each having a reliability R. Let us compute the probability of exactly one out of three components failing, assuming that the failures of the individual components are independent. By rule (3.2), the probability that a single component fails is $1-R$. Then, by rule (3.5), the probability that a single component fails and the other two remain operational is $(1-R)R^2$. Since, the probabilities of any of the three components to fail are the same, then the overall probability of one component failing and other two not is $3(1-R)R^2$. The three probabilities are added by applying rule (3.8), because the events are mutually exclusive. Suppose that one event, A is dependent on another event, B. Then $P(A|B)$ denotes the conditional probability of event A, given event B.

3. Common measures of dependability

In this section, we describe common dependability measures: failure rate, mean time to failure, mean time to repair, mean time between failures and fault coverage.

3.1 Failure rate

Failure rate λ is the expected number of failures per unit time. For example, if a processor fails, on average, once every 1000 hours, then it has a failure rate $\lambda = 1/1000$ failures/hour.

Often failure rate data is available at component level, but not for the entire system. This is because several professional organizations collect and publish failure rate estimates for frequently used components (diodes, switches, gates, flip-flops, etc.). At the same time the design of a new system may involve new configurations of such standard components. When component failure rates are available, a crude estimation of the failure rate of a non-redundant system can be done by adding the failure rates λ_i of the components:

$$\lambda = \sum_{i=1}^{n} \lambda_i$$

Failure rate changes as a function of time. For hardware, a typical evolution of failure rate over a system's life-time is characterized by the phases of infant mortality (I), useful life (II) and wear-out (III). These phases are illustrated by *bathtub curve* relationship shown in Figure 3.1. Failure rate at first decreases due to frequent failures in weak components with manufacturing defects overlooked during manufacturer's testing (poor soldering, leaking capacitor, etc.), then stabilizes after a certain time and then increases as electronic or mechanical components of the system physically wear out.

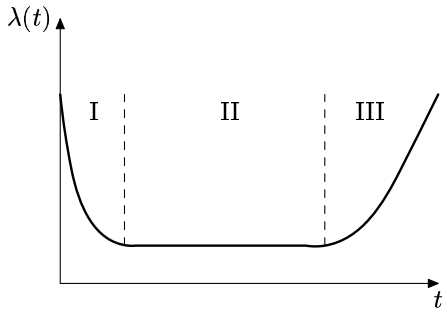

Figure 3.1. Typical evolution of failure rate over a life-time of a hardware system.

During the useful life phase of the system, failure rate function is assumed to have a constant value λ. Then, the reliability of the system varies exponentially as a function of time:

$$R(t) = e^{-\lambda t} \tag{3.9}$$

This law is known as *exponential failure law*. The plot of reliability as a function of time is shown in Figure 3.2.

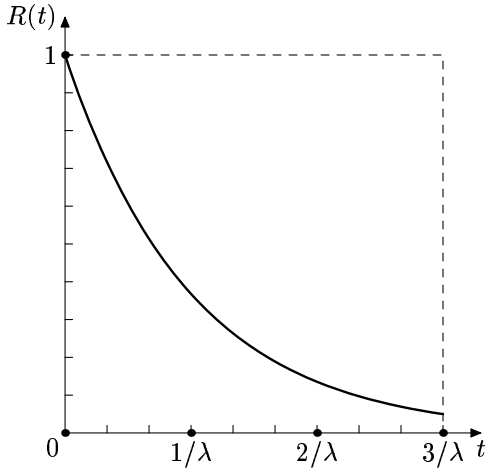

Figure 3.2. Reliability plot $R(t) = e^{-\lambda t}$.

The exponential failure law is very valuable for the analysis of reliability of components and systems in hardware. However, it can only be used in cases when the assumption that the failure rate is constant is adequate. Software failure rate usually decreases as a function of time. A possible curve is shown in Figure 3.3. The three phases of evolution are: test/debug (I), useful life (II) and obsolescence (III).

Software failure rate during useful life depends on the following factors:

1 software process used to develop the design and code

2 complexity of software,

3 size of software,

4 experience of the development team,

5 percentage of code reused from a previous stable project,

6 rigor and depth of testing at test/debug (I) phase.

There are two major differences between hardware and software curves. One difference is that, in the useful-life phase, software normally experiences an increase in failure rate each time a feature upgrade is made. Since the functionality is enhanced by an upgrade, the complexity of software is likely to be increased, increasing the probability of faults. After the increase in failure

rate due to an upgrade, the failure rate levels off gradually, partly because of the bugs found and fixed after the upgrades. The second difference is that, in the last phase, software does not have an increasing failure rate as hardware does. In this phase, the software is approaching obsolescence and there is no motivation for more upgrades or changes.

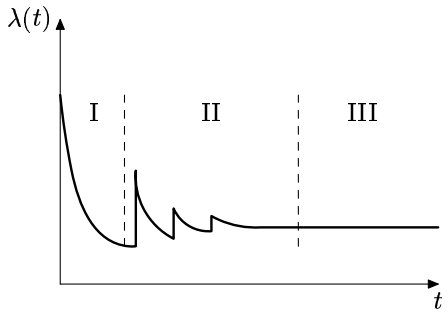

Figure 3.3. Typical evolution of failure rate over a life-time of a software system.

3.2 Mean time to failure

Another important and frequently used measure of interest is mean time to failure defined as follows.

The *mean time to failure* (MTTF) of a system is the expected time of the occurrence of the first system failure.

If n identical systems are placed into operation at time $t = 0$ and the time t_i, $i = \{1, 2, \ldots, n\}$, that each system i operates before failing is measured then the average time is MTTF:

$$MTTF = \frac{1}{n} \cdot \sum_{i=1}^{n} t_i \tag{3.10}$$

In terms of system reliability $R(t)$, MTTF is defined as

$$MTTF = \int_0^\infty R(t)dt. \tag{3.11}$$

So, MTTF is the area under the reliability curve in Figure 3.2. If the reliability function obeys the exponential failure law (3.9), then the solution of (3.11) is given by

$$MTTF = \frac{1}{\lambda} \tag{3.12}$$

where λ is the failure rate of the system. The smaller the failure rate is, the longer is the time to the first failure.

In general, MTTF is meaningful only for systems that operate without repair until they experience a system failure. In a real situation, most of the mission critical systems undergo a complete check-out before the next mission is undertaken. All failed redundant components are replaced and the system is returned to a fully operational status. When evaluating the reliability of such systems, mission time rather than MTTF is used.

3.3 Mean time to repair

The *mean time to repair* (MTTR) of a system is the average time required to repair the system.

MTTR is commonly specified in terms of the *repair rate* μ, which is the expected number of repairs per unit time:

$$MTTR = \frac{1}{\mu} \tag{3.13}$$

MTTR depends on fault recovery mechanism used in the system, location of the system, location of spare modules (on-site versus off-site), maintenance schedule, etc. A low MTTR requirement means a high operational cost of the system. For example, if repair is done by replacing the hardware module, the hardware spares are kept on-site and the site is maintained 24 hours a day, then the expected MTTR can be 30 min. However, if the site maintenance is relaxed to regular working hours on week days only, the expected MTTR increases to 3 days. If the system is remotely located and the operator need to be flown in to replace the faulty module, the MTTR can be 2 weeks. In software, if the failure is detected by watchdog timers and the processor automatically restart the failed tasks, without operating system reboot, then MTTR can be 30 sec. If software fault detection is not supported and a manual reboot by an operator is required, then the MTTR can range from 30 min to 2 weeks, depending on the location of the system.

If the system experiences n failures during its lifetime, the total time that the system is operational is $n \cdot MTTF$. Likewise, the total time the system is being repaired is $n \cdot MTTR$. The steady state availability given by the expression (2.2) can be approximated as

$$A(\infty) = \frac{n \cdot MTTF}{n \cdot MTTF + n \cdot MTTR} = \frac{MTTF}{MTTF + MTTR} \tag{3.14}$$

In section 5.2.2, we will see an alternative approach for computing availability, which uses Markov processes.

3.4 Mean time between failures

The *mean time between failures* (MTBF) of a system is the average time between failures of the system.

If we assume that a repair makes the system perfect, then the relationship between MTBF and MTTF is as follows:

$$MTBF = MTTF + MTTR \tag{3.15}$$

3.5 Fault coverage

There are several types of fault coverage, depending on whether we are concerned with fault detection, fault location, fault containment or fault recovery. Intuitively, fault coverage is the probability that the system will not fail to perform the expected actions when a fault occurs. More precisely, fault coverage is defined in terms of the conditional probability $P(A|B)$, read as "probability of A given B".

Fault detection coverage is the conditional probability that, given the existence of a fault, the system detects it.

$$C = P(\text{fault detection}|\text{fault existence})$$

For example, a system requirement can be that 99% of all single stuck-at faults are detected. The fault detection coverage is a measure of the system's ability to meet such a requirement.

Fault location coverage is the conditional probability that, given the existence of a fault, the system locates it.

$$C = P(\text{fault location}|\text{fault existence})$$

It is common to require systems to locate faults within easily replaceable modules. In this case, the fault location coverage can be used as a measure of success.

Similarly, *fault containment coverage* is the conditional probability that, given the existence of a fault, the system contains it.

$$C = P(\text{fault containment}|\text{fault existence})$$

Finally, *fault recovery coverage* is the conditional probability that, given the existence of a fault, the system recovers.

$$C = P(\text{fault recovery}|\text{fault existence})$$

4. Dependability model types

In this section we consider two common dependability models: reliability block diagrams and Markov processes. Reliability block diagrams belong to a class of *combinatorial models*, which assume that the failures of the individual components are mutually independent. Markov processes belong to a class of *stochastic processes* which take the dependencies between the component failures into account, making the analysis of more complex scenarios possible.

4.1 Reliability block diagrams

Combinatorial reliability models include reliability block diagrams, fault trees, success trees and reliability graphs. In this section we will consider the oldest and most common reliability model: reliability block diagrams.

A reliability block diagram presents an abstract view of the system. The components are represented as blocks. The interconnections among the blocks show the operational dependency between the components. Blocks are connected in series if all of them are necessary for the system to be operational. Blocks are connected in parallel if only one of them is sufficient for the system to operate correctly. A diagram for a two-component serial system is shown in Figure 3.4(a). Figure 3.4(b) shows a diagram of a two-component parallel system. Models of more complex systems may be built by combining the serial and parallel reliability models.

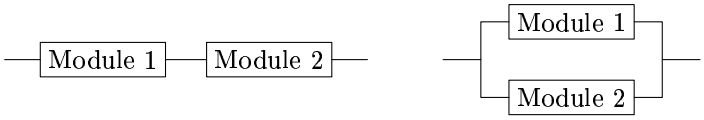

Figure 3.4. Reliability block diagram of a two-component system: (a) serial, (b) parallel.

As an example, consider a system consisting of two duplicated processors and a memory. The reliability block diagram for this system is shown in Figure 3.5. The processors are connected in parallel, since only one of them is sufficient for the system to be operational. The memory is connected in series, since its failure would cause the system failure.

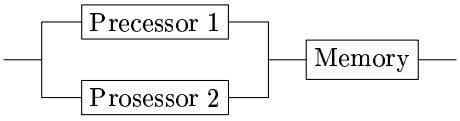

Figure 3.5. Reliability block diagram of a three-component system.

Reliability block diagrams are a popular model, because they are easy to understand and to use for modeling systems with redundancy. In the next section we will see that they are also easy to evaluate using analytical methods. However, reliability block diagrams, as well as other combinatorial reliability models, have a number of serious limitations.

First, reliability block diagrams assume that the system components are limited to the operational and failed states and that the system configuration does not change during the mission. Hence, they cannot model standby components, repair, as well as complex fault detection and recovery mechanisms. Second, the failures of the individual components are assumed to be independent. Therefore, the case when the sequence of component failures affects system reliability cannot be adequately represented.

4.2 Markov processes

Contrary to combinatorial models, Markov processes take into account the interactions of component failures making the analysis of complex scenarios possible. Markov processes theory derives its name from the Russian mathematician A. A. Markov (1856-1922), who pioneered a systematic investigation of describing random processes mathematically.

Markov processes are a special class of stochastic processes. The basic assumption is that the behavior of the system in each state is memoryless. The transition from the current state of the system is determined only by the present state and not by the previous state or the time at which it reached the present state. Before a transition occurs, the time spent in each state follows an exponential distribution. In dependability engineering, this assumption is satisfied if all events (failures, repairs, etc.) in each state occur with constant occurrence rates.

Markov processes are classified based on state space and time space characteristics as shown in Table 3.1. In most dependability analysis applications,

State Space	Time Space	Common Model Name
Discrete	Discrete	Discrete Time Markov Chains
Discrete	Continuous	Continuous Time Markov Chains
Continuous	Discrete	Continuous State, Discrete Time Markov Processes
Continuous	Continuous	Continuous State, Continuous Time Markov Processes

Table 3.1. Four types of Markov processes.

the state space is discrete. For example, a system might have two states: operational and failed. The time scale is usually continuous, which means that component failure and repair times are random variables. Thus, *Continuous Time Markov Chains* are the most commonly used. In some textbooks, they are called *Continuous Markov Models*. There are, however, applications in which time scale is discrete. Examples include synchronous communication protocol, shifts in equipment operation, etc. If both time and state space are discrete, then the process is called *Discrete Time Markov Chain*.

Markov processes are illustrated graphically by state transition diagrams. A *state transition diagram* is a directed graph $G = (V, E)$, where V is the set of vertices representing *system states* and E is the set of edges representing *system transitions*. State transition diagram is a mathematical model which can be used to represent a wide variety of processes, i.e. radioactive breakdown or chemical reaction. For dependability models, a state is defined to be a particular combination of operating and failed components. For example, if we have a system consisting of two components, then there are four different combinations enumerated in Table 3.2, where O indicates an operational component and F indicates a failed component.

Component		State
1	2	Number
O	O	1
O	F	2
F	O	3
F	F	4

Table 3.2. Markov states of a two-component system.

The state transitions reflect the changes which occur within the system state. For example, if a system with two identical component is in state (11), and the first module fails, then the system moves to state (01). So, a Markov process represents possible chains of events which occur within a system. In the case of dependability analysis, these events are failures and repairs.

Each edge carries a label, reflecting the rate at which the state transitions occur. Depending on the modeling goals, this can be failure rate, repair rate or both.

We illustrate the concept first on a simple system, consisting of a single component.

4.2.1 Single-component system

A single component has only two states: one operational (state 1) and one failed (state 2). If no repair is allowed, there is a single, non-reversible transition between the states, with a label λ corresponding to the failure rate of the component (Figure 3.6).

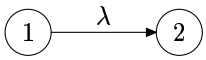

Figure 3.6. State transition diagram of a single-component system.

If repair is allowed, then a transition between the failed and the operational states is possible, with a repair rate μ (Figure 3.7). State diagrams incorporating repair are used in availability analysis.

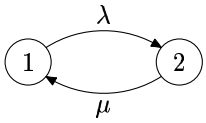

Figure 3.7. State transition diagram of a single-component system incorporating repair.

Next, suppose that we would like to distinguish between failed-safe and failed-unsafe states, as required in safety analysis. Let state 2 be a failed-safe and state 3 be a fail-unsafe state (Figure 3.8). The transition between state 1 and state 2 depends on both component failure rate λ and the probability that, is a fault exists, the system succeeds in detecting it and in taking the corresponding actions to fail in a safe manner, i.e. on fault coverage C. The transition between state 1 and the failed-unsafe state 3 depends on the failure rate λ and the probability that a fault is *not* detected, i.e. $1 - C$.

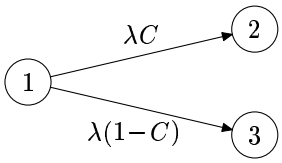

Figure 3.8. State transition diagram of a single-component system for safety analysis.

4.2.2 Two-component system

A two-component system has four possible states, enumerated in Table 3.2. The changes of states are illustrated by a state transition diagram shown in

Figure 3.9. The failure rates λ_1 and λ_2 for components 1 and 2 indicate the rates at which the transitions are made between the states. The two components are assumed to be independent and non-repairable.

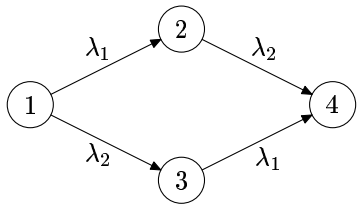

Figure 3.9. State transition diagram for a two independent component system.

If the components are in a serial configuration, then any component failure causes system failure. So, only state 1 is an operational state. States 2, 3 and 4 are failed states. If the components are in parallel, both components must fail to have a system failure. Therefore, states 1, 2 and 3 are the operational states, whereas state 4 is a failed state.

4.2.3 State transition diagram simplification

It is often possible to reduce the size of a state transition diagram without a sacrifice in accuracy. For example, suppose the components in the two-component system shown in Figure 3.9 are in parallel. If the components have identical failure rates $\lambda_1 = \lambda_2 = \lambda$, then it is not necessary to distinguish between the states 2 and 3. Both states represent a condition where one component is operational and one is failed. So, we can merge these two states into one. (Figure 3.10). The assignments of the state numbers in the simplified transition diagram are shown in Table 3.3. Since the failures of components are assumed

Component		State
1	2	Number
O	*O*	1
O	*F*	2
F	*O*	2
F	*F*	3

Table 3.3. Markov states of a simplified state transition diagram of a two-component parallel system.

to be independent events, the transition rate from state 1 to state 2 in Figure

3.10 is the sum of the transition rates from state 1 to states 2 and 3 in Figure 3.9, i. e. 2λ.

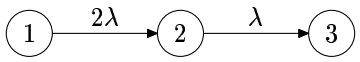

Figure 3.10. Simplified state transition diagram of a two-component parallel system.

5. Dependability computation methods

In this section we study how reliability block diagrams and Markov processes can be used to evaluate system dependability.

5.1 Computation using reliability block diagrams

Reliability block diagrams can be used to compute system reliability as well as system availability.

5.1.1 Reliability computation

To compute the reliability of a system represented by a reliability block diagram, we need first to break the system down into its serial and parallel parts. Next, the reliabilities of these parts are computed. Finally, the overall solution is composed from the reliabilities of the parts.

Given a system consisting of n components with $R_i(t)$ being the reliability of the ith component, the reliability of the overall system is given by

$$R(t) = \begin{cases} \prod_{i=1}^{n} R_i(t) & \text{for a series structure,} \\ 1 - \prod_{i=1}^{n} (1 - R_i(t)) & \text{for a parallel structure.} \end{cases} \tag{3.16}$$

In a serial system, all components should be operational for a system to function correctly. Hence, by rule (3.5), $R_{serial}(t) = \prod_{i=1}^{n} R_i(t)$. In a parallel system, only one of the components is required for a system to be operational. So, the unreliability of a parallel system is equal to the probability that all n elements fail, i.e. $Q_{parallel}(t) = \prod_{i=1}^{n} Q_i(t) = \prod_{i=1}^{n} (1 - R_i(t))$. Hence, by rule 1, $R_{parallel}(t) = 1 - Q_{parallel}(t) = 1 - \prod_{i=1}^{n} (1 - R_i(t))$.

Designing a reliable serial system is difficult. For example, if a serial system with 100 components is to be build, and each of the components has a reliability 0.999, the overall system reliability is $0.999^{100} = 0.905$.

On the other hand, a parallel system can be made reliable despite the unreliability of its component parts. For example, a parallel system of four identical modules with the module reliability 0.95, has the system reliabil-

ity $1-(1-.95)^4 = 0.99999375$. Clearly, however, the cost of the parallelism can be high.

5.1.2 Availability computation

If we assume that the failure and repair times are independent, then we can use reliability block diagrams to compute the system availability. This situation occurs when the system has enough spare resources to repair all the failed components simultaneously. Given a system consisting of n components with $A_i(t)$ being the availability of the ith component, the availability if the overall system is given by

$$A(t) = \begin{cases} \prod_{i=1}^{n} A_i(t) & \text{for a series structure,} \\ 1 - \prod_{i=1}^{n} (1 - A_i(t)) & \text{for a parallel structure.} \end{cases} \tag{3.17}$$

The combined availability of two components in series is always lower than the availability of the individual components. For example, if one component has the availability 99% (3.65 days/year downtime) and another component has the availability 99.99% (52 minutes/year downtime), then the availability of the system consisting of these two components in series is 98.99% (3.69 days/year downtime). On the contrary, a parallel system consisting of three identical components with individual availability 99% has an availability of 99.9999 (31 seconds/year downtime).

5.2 Computation using Markov processes

In this section we show how Markov processes are used to evaluate system dependability. Continuous Time Markov Chains are the most important class of Markov processes for dependability analysis, so the presentation is focused on this model.

The aim of Markov processes analysis is to calculate $P_i(t)$, the probability that the system is in state i at time t. Once this is known, the system reliability, availability or safety can be computed as a sum taken over all the operating states.

Let us designate state 1 as the state in which all the components are operational. Assuming that at $t = 0$ the system is in state 1, we get

$$P_1(0) = 1.$$

Since at any time the system can be only in one state, $P_i(0) = 0, \forall i \neq 1$, and we have

$$\sum_{i \in O \cup F} P_i(t) = 1, \tag{3.18}$$

where the sum is over all possible states.

To determine the $P_i(t)$, we derive a set of differential equations, one for each state of the system. These equations are called *state transition equations* because they allow the $P_i(t)$ to be determined in terms of the rates (failure, repair) at which transitions are made from one state to another. State transition equations are usually presented in matrix form. The matrix M whose entry m_{ij} is the rate of transition between the states i and j is called the *transition matrix* associated with the system. We use the first index i for the columns of the matrix and the second index j for the rows, i.e. M has the following structure

$$\mathbf{M} = \begin{bmatrix} m_{11} & m_{21} & \dots & m_{k1} \\ m_{12} & m_{22} & \dots & m_{k1} \\ & & \dots & \\ m_{1k} & m_{2k} & \dots & m_{kk} \end{bmatrix}.$$

where k is the number of states in the state transition diagram representing the system. In reliability or availability analysis the components of the system are normally assumed to be in either operational or failed states. So, if a system consists of n components, then $k \leq 2^n$. In safety analysis, where the system can fail in either a safe or an unsafe way, k can be up to 3^n. The entries in each column of the transition matrix must sum up to 0. So, the entries m_{ii} corresponding to self-transitions are computed as $-\sum m_{ij}$, for all $j \in \{1,2,\dots k\}$ such that $j \neq i$.

For example, the transition matrix for the state transition diagram of a single-component system shown in Figure 3.6 is

$$\mathbf{M} = \begin{bmatrix} -\lambda & 0 \\ \lambda & 0 \end{bmatrix}. \tag{3.19}$$

The rate of transition between the states 1 and 2 is λ, therefore the $m_{12} = \lambda$. Hence, $m_{11} = -\lambda$. The rate of transition between the states 2 and 1 is 0, so $m_{21} = 0$ and thus $m_{22} = 0$.

Similarly, the transition matrix for the state transition diagram in Figure 3.7, which incorporates repair, is

$$\mathbf{M} = \begin{bmatrix} -\lambda & \mu \\ \lambda & -\mu \end{bmatrix}. \tag{3.20}$$

The transition matrix for the state transition diagram in Figure 3.8, is of size 3×3, since, for safety analysis, the system is modeled to be in three different states: operational, failed-safe failed-unsafe.

$$\mathbf{M} = \begin{bmatrix} -\lambda & 0 & 0 \\ \lambda C & 0 & 0 \\ \lambda(1-C) & 0 & 0 \end{bmatrix}. \tag{3.21}$$

The transition matrix for the simplified state transition diagram of the two-component system, shown in Figure 3.10 is

$$\mathbf{M} = \begin{bmatrix} -2\lambda & 0 & 0 \\ 2\lambda & -\lambda & 0 \\ 0 & \lambda & 0 \end{bmatrix}. \tag{3.22}$$

The examples above illustrate two important properties of transition matrices. One, which we have mentioned before, is that the sum of the entries in each column is zero. Positive sign of an $ij$the entry indicates that the transition originates in the ith state. Negative sign of an $ij$the entry indicates that the transition terminates in the ith state.

Second property of the transition matrix is that it allows us to distinguish between the operational and failed states. In reliability analysis, once a system fails, the failed state cannot be leaved. Therefore, each failed state i has a zero diagonal element m_{ii}. This is not the case, however, when availability or safety are computed, as one can see from (3.20) and (3.21).

Using state transition matrices, state transition equations are derived as follows. Let $\mathbf{P}(t)$ be a vector whose ith element is the probability $P_i(t)$ that the system is in state i at time t. Then the matrix representation of a system of state transition equations is given by

$$\frac{d}{dt}\mathbf{P}(t) = \mathbf{M} \cdot \mathbf{P}(t). \tag{3.23}$$

Once the system of equations is solved and the $P_i(t)$ are known, the system's reliability, availability or safety can be computed as a sum taken over all the operating states.

We illustrate the computation process on a number of simple examples.

5.2.1 Reliability evaluation

Independent components case

Let us first compute the reliability of a parallel system consisting of two independent components which we have considered before (Figure 3.9). Applying (3.23) to the matrix (3.22) we get

$$\frac{d}{dt}\begin{bmatrix} P_1(t) \\ P_2(t) \\ P_3(t) \end{bmatrix} = \begin{bmatrix} -2\lambda & 0 & 0 \\ 2\lambda & -\lambda & 0 \\ 0 & \lambda & 0 \end{bmatrix} \cdot \begin{bmatrix} P_1(t) \\ P_2(t) \\ P_3(t) \end{bmatrix}.$$

The above matrix form represents the following system of state transition equations

$$\begin{cases} \frac{d}{dt}P_1(t) = -2\lambda P_1(t) \\ \frac{d}{dt}P_2(t) = 2\lambda P_1(t) - \lambda P_2(t) \\ \frac{d}{dt}P_3(t) = \lambda P_2(t) \end{cases}$$

By solving this system of equations, we get

$$P_1(t) = e^{-2\lambda t}$$
$$P_2(t) = 2e^{-\lambda t} - 2e^{-2\lambda t}$$
$$P_3(t) = 1 - 2e^{-\lambda t} + e^{-2\lambda t}$$

Since the $P_i(t)$ are known, we can now calculate the reliability of the system. For the parallel configuration, both components should fail to have a system failure. Therefore, the reliability of the system is the sum of probabilities $P_1(t)$ and $P_2(t)$:

$$R_{parallel}(t) = 2e^{-\lambda t} - e^{-2\lambda t} \tag{3.24}$$

In general case, the reliability of the system is computed as a function using the equation

$$R(t) = \sum_{i \in O} P_i(t), \tag{3.25}$$

where the sum is taken over all the operating states O. Alternatively, the reliability can be calculated

$$R(t) = 1 - \sum_{i \in F} P_i(t),$$

where the sum is taken over all the states F in which the system has failed.

Note that, for constant failure rates, the component reliability is $R(t) = e^{-\lambda t}$. Therefore, the equation (3.24) can be written as $R_{parallel}(t) = 2R^2 - R$, which agrees with the expression (3.16) derived using reliability block diagrams. Two results are the same, because in this example we assumed the failure rates to be mutually independent.

Dependant components case

The value of Markov processes becomes evident in situations in which component failure rates are no longer assumed to be independent of the system state. One of the common cases of dependence is load-sharing components, which we consider next. Another possibility is the case of standby components, which is considered in the availability computation section.

The word *load* is used in a broad sense of the stress on a system. This can be an electrical load, a load caused by high temperature, or an information load. In practice, failure rates are found to increase with loading. Suppose that two components share a load. If one of the component fails, the additional load on the second component is likely to increase its failure rate.

To model load-sharing failures, consider the state transition diagram of a two-component parallel system shown in Figure 3.11. As before, we have four states. However, after one component failure, the failure rate of the second component increases. The increased failure rates of the components 1 and 2 are denoted with λ_1' and λ_2', respectively.

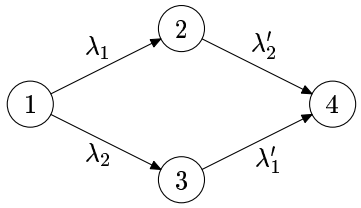

Figure 3.11. State transition diagram of a two-component parallel system with load sharing.

From the state transition diagram in Figure 3.11, we can derive the state transition equations for $P_i(t)$. In the matrix form they are

$$\frac{d}{dt}\begin{bmatrix} P_1(t) \\ P_2(t) \\ P_3(t) \\ P_4(t) \end{bmatrix} = \begin{bmatrix} -\lambda_1-\lambda_2 & 0 & 0 & 0 \\ \lambda_1 & -\lambda_2' & 0 & 0 \\ \lambda_2 & 0 & -\lambda_1' & 0 \\ 0 & \lambda_2' & \lambda_1' & 0 \end{bmatrix} \cdot \begin{bmatrix} P_1(t) \\ P_2(t) \\ P_3(t) \\ P_4(t) \end{bmatrix}.$$

By expanding the matrix form, we get the following system of equations

$$\begin{cases} \frac{d}{dt}P_1(t) = (-\lambda_1-\lambda_2)P_1(t) \\ \frac{d}{dt}P_2(t) = \lambda_1 P_1(t) - \lambda_2' P_2(t) \\ \frac{d}{dt}P_3(t) = \lambda_2 P_1(t) - \lambda_1' P_3(t) \\ \frac{d}{dt}P_4(t) = \lambda_2' P_2(t) + \lambda_1' P_3(t). \end{cases}$$

The solution of this system of equation is

$$\begin{aligned} P_1(t) &= e^{(-\lambda_1-\lambda_2)t} \\ P_2(t) &= \tfrac{\lambda_1}{\lambda_1+\lambda_2-\lambda_2'}e^{\lambda_2' t} - \tfrac{\lambda_1}{\lambda_1+\lambda_2-\lambda_2'}e^{(-\lambda_1-\lambda_2)t} \\ P_3(t) &= \tfrac{\lambda_2}{\lambda_1+\lambda_2-\lambda_1'}e^{\lambda_1' t} - \tfrac{\lambda_2}{\lambda_1+\lambda_2-\lambda_1'}e^{(-\lambda_1-\lambda_2)t} \\ P_4(t) &= 1 - e^{(-\lambda_1-\lambda_2)t} - \tfrac{\lambda_1}{\lambda_1+\lambda_2-\lambda_2'}e^{\lambda_2' t} + \tfrac{\lambda_1}{\lambda_1+\lambda_2-\lambda_2'}e^{(-\lambda_1-\lambda_2)t} \\ &\quad - \tfrac{\lambda_2}{\lambda_1+\lambda_2-\lambda_1'}e^{\lambda_1' t} + \tfrac{\lambda_2}{\lambda_1+\lambda_2-\lambda_1'}e^{(-\lambda_1-\lambda_2)t}. \end{aligned}$$

Finally, since both components should fail for the system to fail, the reliability is equal to $1 - P_4(t)$, yielding the expression

$$\begin{aligned} R_{parallel}(t) = {} & e^{(-\lambda_1-\lambda_2)t} + \tfrac{\lambda_1}{\lambda_1+\lambda_2-\lambda_2'}e^{\lambda_2' t} - \tfrac{\lambda_1}{\lambda_1+\lambda_2-\lambda_2'}e^{(-\lambda_1-\lambda_2)t} \\ & + \tfrac{\lambda_2}{\lambda_1+\lambda_2-\lambda_1'}e^{\lambda_1' t} - \tfrac{\lambda_2}{\lambda_1+\lambda_2-\lambda_1'}e^{(-\lambda_1-\lambda_2)t}. \end{aligned} \tag{3.26}$$

If $\lambda_1' = \lambda_1$ and $\lambda_2' = \lambda_2$, the above equation is equal to (3.24). The effect of the increased loading can be illustrated as follows. Assume that the two components are identical, i.e. $\lambda_1 = \lambda_2 = \lambda$ and $\lambda_1' = \lambda_2' = \lambda'$. Then, the equation (3.26) reduces to

$$R_{parallel}(t) = \frac{2\lambda}{2\lambda - \lambda'} e^{-\lambda' t} - \frac{\lambda'}{2\lambda - \lambda'} e^{-2\lambda t}.$$

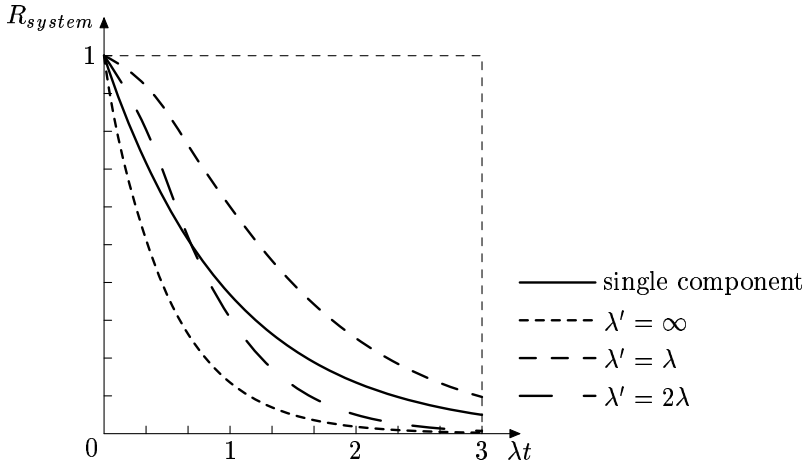

Figure 3.12. Reliability of a two-component parallel system with load sharing.

Figure 3.12 shows the reliability of a two-component parallel system with load-sharing for different values of λ'. The reliability $e^{-\lambda t}$ of a single-component system is also plotted for a comparison. In the case where $\lambda' = \lambda$ two two components are independent, so the reliability is given by (3.23). $\lambda' = \infty$ is the case of total dependency. This means that the failure of one component would bring the immediate failure of the other component. So, in this case, the reliability will be equal to the reliability of a serial system with two components (3.16). It can been seen that, the more the value of λ' exceeds the value of λ, the closer the reliability of this system approaches the reliability of the serial system with two components.

5.2.2 Availability evaluation

In availability analysis, as well as in reliability analysis, there are situations in which the component failures cannot be considered independent of one another. These include shared-load systems and systems with standby components, which are repairable.

The dependencies between component failures can be analyzed using Markov methods, provided that the failures are detected and that the failure and repair

rates are time-independent. There is a fundamental difference between treatment of repair for reliability and availability analysis. In reliability calculations, components are allowed to be repaired only as long as the system has not failed. In availability calculations, the components can also be repaired after the system failure.

The difference is best illustrated on a simple example of a system with two components, one primary and one standby. The standby component is held in reserve and only brought to operation when the primary component fails. We assume that there is a perfect fault detection unit which detects a failure in the primary component and replaces it by the standby component. We also assume that the standby component cannot fail while it is in the standby mode.

The state transition diagrams of the standby system for reliability and availability analysis are shown in Figure 3.13(a) and (b), respectively. The states are numbered according to the Table 3.4. When the primary component fails,

Component		State
1	2	Number
O	*O*	1
F	*O*	2
F	*F*	3

Table 3.4. Markov states of a simplified state transition diagram of a two-component parallel system incorporating repair.

there is a transition between states 1 and 2. If the system is in state 2 and the backup component fails, there is a transition to state 3. Since we assumed that the backup unit cannot fail while in the standby mode, the combination (O,F) cannot occur. States 1 and 2 are operational states. State 3 is the failed state.

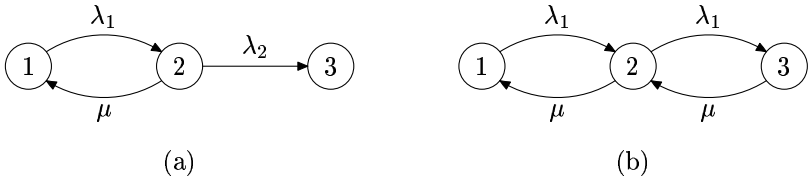

Figure 3.13. Sate transition diagrams for a standby two-component system (a) for reliability analysis, (b) for availability analysis.

Suppose the primary unit can be repaired with a rate μ. For reliability analysis, this implies that a transition between states 2 and 1 is possible. The

corresponding transition matrix if given by

$$\mathbf{M} = \begin{bmatrix} -\lambda_1 & \mu & 0 \\ \lambda_1 & -\lambda_2 - \mu & 0 \\ 0 & \lambda_2 & 0 \end{bmatrix}.$$

For availability analysis, we should be able to repair the backup unit as well. This adds a transition between states 3 and 2. If we assume that the repair rates for the primary and the backup units are the same, and also that the backup unit is repaired first, then the corresponding transition matrix if given by

$$\mathbf{M} = \begin{bmatrix} -\lambda_1 & \mu & 0 \\ \lambda_1 & -\lambda_2 - \mu & \mu \\ 0 & \lambda_2 & -\mu \end{bmatrix}. \tag{3.27}$$

One can see that, in the matrix for availability calculations, none of the diagonal elements is zero. This is because the system should be able to recover from the failed state. By solving the system of state transition equations, we can get $P_i(t)$ and compute the availability of the system as

$$A(t) = 1 - \sum_{i \in F} P_i(t), \tag{3.28}$$

where the sum is taken over all the failed states F.

Usually, the steady state availability rather than the time dependent availability is of interest. The steady state availability can be computed in a simpler way. We note that, as time approaches infinity, the derivative on the right-hand side of the equation 3.23 vanishes and we get a time-independent relationship

$$\mathbf{M} \cdot \mathbf{P}(\infty) = 0. \tag{3.29}$$

In our example, for matrix (3.27) this represents a system of equations

$$\begin{cases} -\lambda_1 P_1(\infty) + \mu P_2(\infty) = 0 \\ \lambda_1 P_1(\infty) - (\lambda_2 + \mu) P_2(\infty) + \mu P_3(\infty) = 0 \\ \lambda_2 P_2(\infty) - \mu P_3(\infty) = 0 \end{cases}$$

Since these three equations are linearly dependent, they are not sufficient to solve for $P(\infty)$. The needed piece of additional information is the condition (3.18) that the sum of all probabilities is one:

$$\sum_i P_i(\infty) = 1. \tag{3.30}$$

If we assume $\lambda_1 = \lambda_2 = \lambda$, then we get

$$P_1(\infty) = \left[1 + \tfrac{\lambda}{\mu} + (\tfrac{\lambda}{\mu})^2\right]^{-1},$$
$$P_2(\infty) = \left[1 + \tfrac{\lambda}{\mu} + (\tfrac{\lambda}{\mu})^2\right]^{-1} \tfrac{\lambda}{\mu},$$
$$P_3(\infty) = \left[1 + \tfrac{\lambda}{\mu} + (\tfrac{\lambda}{\mu})^2\right]^{-1} \left(\tfrac{\lambda}{\mu}\right)^2.$$

The steady-state availability can be found by setting $t = \infty$ in (3.28)

$$A(\infty) = 1 - \left[1 + \frac{\lambda}{\mu} + (\frac{\lambda}{\mu})^2\right]^{-1} \left(\frac{\lambda}{\mu}\right)^2.$$

If we further assume that $\lambda/\mu << 1$, we can write

$$A(\infty) \approx 1 - \left(\frac{\lambda}{\mu}\right)^2.$$

To summarize, steady-state availability problems are solved by the same procedure as time-dependent availability. Any $n-1$ of the n equations represented by (3.29) are combined with the condition 3.30 to solve for the components of $\mathbf{P}(\infty)$. These are then substituted into (3.28) to obtain availability.

5.2.3 Safety evaluation

The main difference between safety calculation and reliability calculation is in the construction of the state transition diagram. As we mentioned before, for safety analysis, the failed state is splitted into failed-safe and failed-unsafe ones. Once the state transition diagram for a system is derived, the state transition equations are obtained and solved using same procedure as for reliability analysis.

As an example, consider the single component system shown in Figure 3.8. Its state transition matrix is given by (3.21). So, the state transition equations for $P_i(t)$ are given by

$$\frac{d}{dt}\begin{bmatrix} P_1(t) \\ P_2(t) \\ P_3(t) \end{bmatrix} = \begin{bmatrix} -\lambda & 0 & 0 \\ \lambda C & 0 & 0 \\ \lambda(1-C) & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} P_1(t) \\ P_2(t) \\ P_3(t) \end{bmatrix}.$$

The solution of this system of equations is

$$P_1(t) = e^{-\lambda t}$$
$$P_2(t) = C - Ce^{-\lambda t}$$
$$P_3(t) = (1-C) - (1-C)e^{-\lambda t}$$

The safety of the system is the sum of probabilities of being in the operational or fail-safe states, i.e.

$$S(t) = P_1(t) + P_2(t) = C + (1-C)e^{-\lambda t}$$

At time $t = 0$ the safety of the system is 1, as expected. As time approaches infinity, the safety approaches the fault detection coverage, $S(0) = C$. So, if $C = 1$, the system has a perfect safety.

6. Problems

3.1. Why is dependability evaluation important?

3.2. What is the difference between qualitative and quantitative evaluation?

3.3. Define the failure rate. How can the failure rate of a non-redundant system can be computed from the failure rates of its components?

3.4. How does a typical evolution of failure rate over a system's life-time differ for hardware and software?

3.5. What is the mean time to failure of a system? How can the MTTF of a non-redundant system be computed from the MTTF of its components?

3.6. What is the difference between the mean time to repair and the mean time between failures?

3.7. A heart pace-maker has a constant failure rate of $\lambda = 8.167/10^9$ hr.

(a) What is the probability that it will fail during the first year of operation?

(b) What is the probability that it will fail within 5 years of operation?

(b) What is the MTTF?

3.8. A logic circuit with no redundancy consists of 16 two-input NAND gates and 3 J-K flip-flops. Assuming the constant failure rates of a two-input NAND gate and a J-K flip-flop are 0.2107 and 0.4667 per hour, respectively, compute

(a) the failure rate of the logic circuit,

(b) the reliability for a 72-hour mission,

(c) the MTTF.

3.9. An automatic teller machine manufacturer determines that his product has a constant failure rate of $\lambda = 77.16$ per 10^6 hours in normal use. For how long should the warranty be set if no more than 5% of the machines are to be returned to the manufacturer for repair?

3.10. A car manufacturer estimates that the reliability of his product is 99% during the first 7 years.

(a) How many cars will need a repair during the first year?

(b) What is the MTTF?

3.11. A two-year guarantee is given on a TV-set based on the assumption that no more than 3% of the items will be returned for repair. Assuming exponential failure law, what is the maximum failure rate that can be tolerated?

3.12. A DVD-player manufacturer determines that the average DVD set is used 930 hr/year. A two-year warranty is offered on the DVD set having MTTF of 2500 hr. If the exponential failure law holds, which fraction of DVD-sets will fail during the warranty period?

3.13. Suppose the failure rate of a jet engine is $\lambda = 10^{-3}$ per hour. What is the probability that more than two engines on a four-engine aircraft will fail during a 4-hour flight? Assume that the failures are independent.

3.14. A non-redundant system with 50 components has a design life reliability of 0.95. The system is re-designed so that it has only 35 components. Estimate the design life reliability of the re-designed system. Assume that all the components have constant failure rate of the same value and the failures are independent.

3.15. At the end of the year of service the reliability of a component with a constant failure rate is 0.96.

(a) What is the failure rate?

(b) If two components are put in parallel, what is the one year reliability? Assume that the failures are independent.

3.16. A lamp has three 25V bulbs. The failure rate of a bulb is $\lambda = 10^{-3}$ per year. What is the probability that more than one bulb fail during the first month?

3.17. Suppose a component has a failure rate of $\lambda = 0.007$ per hour. How many components should be placed in parallel if the system is to run for 200 hours with failure probability of no more than 0.02? Assume that the failures are independent.

3.18. Consider a system with three identical components with failure rate λ. Find the system failure rate λ_{sys} for the following cases:

(a) All three components in series.

(b) All three components in parallel.

(c) Two components in parallel and the third in series.

3.19. The MTTF of a system with constant failure rate has been determined to be 1000 hr. An engineer is to set the design life time so that the end-of-life reliability is 0.95.

(a) Determine the design life time.

(b) If two systems are placed in parallel, to what value may the design life time be increased without decreasing the end-of-life reliability?

3.20. A printer has an MTTF = 168 hr and MTTR = 3 hr.

(a) What is the steady state availability?

(b) If MTTR is reduced to 1 hr, what MTTF can be tolerated without decreasing the availability of the printer?

3.21. A copy machine has a failure rate of 0.01 per week. What repair rate should be maintained to achieve a steady state availability of 95%?

3.22. Suppose that the steady state availability for standby system should be 0.9. What is the maximum acceptable value of the failure to repair ratio λ/μ?

3.23. A computer system is designed to have a failure rate of one fault in 5 years. The rate remains constant over the life of the system. The system has no fault-tolerance capabilities, so it fails upon occurrence of the first fault. For such a system:

(a) What is the MTTF?

(b) What is the probability that the system will fail in its first year of operation?

(c) The vendor of this system wishes to offer insurance against failure for the three years of operation of the system at some extra cost. The vendor determined that it should charge $200 for each 10% drop in reliability to offer such an insurance. How much should the vendor charge to the client for such an insurance?

3.24. What are the basic assumptions regarding the failures of the individual components (a) in reliability block diagram model; (b) in Markov process model?

3.25. A system consists of three modules: M_1, M_2 and M_3. It was analyzed, and the following reliability expression was derived:

$$R_{system} = R_1R_3 + R_2R_3 - R_1R_2R_3$$

Draw the reliability block diagram for this system.

3.26. A system with four modules: A, B, C and D is connected so that it operates correctly only if one of the two conditions is satisfied:

- modules A and D operate correctly,
- module A operates correctly and either B or C operates correctly.

Answer the following questions:

(a) Draw the reliability block diagram of the above system such that every module appears only once.

(b) What is the reliability of the system? Assume that the reliability of the module X is $R(X)$ and that the modules fail independently from each other.

(c) What is the reliability of the above system as a function of time t, if the failure rates of the components are $\lambda_A = \lambda_B = \lambda_C = \lambda_D = 0.01$ per year? Assume that the exponential failure law holds.

3.27. How many states has a non-simplified state transition diagram of a system consisting of n components? Assume that every component has only two states: operational and failed.

3.28. Construct the Markov model of the three-component system shown in Figure 3.5. Assume that the components are independent and non-repairable. The failure rate of the processors 1 and 2 is λ_p. The failure rates of the memory is λ_m. Derive and solve the system of state transition equations representing this system. Compute the reliability of the system.

3.29. Construct the Markov model of the three-component system shown in Figure 3.5 for the case when a failed processor can be repaired. Assume that the components are independent and that a processor can be repaired as long as the system has not failed. The failure rate of the processors 1 and 2 is λ_p. The failure rate of the memory is λ_m. Derive and solve the system of state transition equations representing this system. Compute the reliability of the system.

3.30. What is the difference between treatment of repair for reliability and availability analysis?

3.31. Construct the Markov model of the three-component system shown in Figure 3.5 for availability analysis. Assume that the components are independent and that the processors and the memory can be repaired after the system failure. The failure rate of the processors 1 and 2 is λ_p. The failure rate of the memory is λ_m. Derive and solve the system of state transition equations representing this system. Compute the reliability of the system.

3.32. Suppose that the reliability of a system consisting of 4 blocks, two of which are identical, is given by the following equation:

$$R_{system} = R_1R_2R_3 + R_1^2 - R_1^2R_2R_3$$

Draw the reliability block diagram representing the system.

3.33. Construct a Markov chain and write a transition matrix for self-purging redundancy with 3 modules, for the cases listed below. For all cases, assume that the component's failures are independent events and that the failure rate of each module is λ. For all cases, simplify state transition diagrams as much as you can.

(a) Do reliability evaluation, assuming that the voter and switches are perfect and no repairs are allowed.

(b) Do reliability evaluation, assuming that the voter can fail with the failure rate λ_v, the switches are perfect, and no repairs are allowed.

(c) Do reliability evaluation, assuming that the voter and switches are perfect and repairs are allowed. Assume that each module has its own repair crew (i.e. that the component's repairs are independent events) and that the repair rate of each module is μ.

(d) Do availability evaluation, assuming that the voter and switches are perfect and repairs are allowed. Assume that there is a single repair crew for all modules that can handle only one module at a time and that the repair rate of each module is μ.

Chapter 4

HARDWARE REDUNDANCY

Those parts of the system that you can hit with a hammer (not advised) are called hardware; those program instructions that you can only curse at are called software.

—Anonymous

1. Introduction

Hardware redundancy is achieved by providing two or more physical instances of a hardware component. For example, a system can include redundant processors, memories, buses or power supplies. Hardware redundancy is often the only available method for improving the dependability of a system, when other techniques, such as better components, design simplification, manufacturing quality control, software debugging, have been exhausted or shown to be more costly than redundancy.

Originally, hardware redundancy techniques were used to cope with the low reliability of individual hardware elements. Designers of early computing systems replicated components at gate and flip-flop levels and used comparison or voting to detect or correct faults. As reliability of hardware components improved, the redundancy was shifted at the level of larger components, such as whole memories or arithmetic units, thus reducing the relative complexity of the voter or comparator with respect to that of redundant units.

There are three basic forms of hardware redundancy: passive, active and hybrid. *Passive redundancy* achieves fault tolerance by masking the faults that occur without requiring any action on the part of the system or an operator. *Active redundancy* requires a fault to be detected before it can be tolerated. After the detection of the fault, the actions of location, containment and recovery are performed to remove the faulty component from the system. *Hybrid redundancy* combine passive and active approaches. Fault masking is used to

prevent generation of erroneous results. Fault detection, location and recovery are used to replace the faulty component with a spare component.

Hardware redundancy brings a number of penalties: increase in weight, size, power consumption, as well as time to design, fabricate, and test. A number of choices need to be examined to determine a best way to incorporate redundancy into the system. For example, weight increase can be reduced by applying redundancy to the lower-level components. Cost increase can be minimized if the expected improvement in dependability reduces the cost of preventive maintenance for the system. In this section, we examine a number of different redundancy configurations and calculate the effect of redundancy on system dependability. We also discuss the problem of common-mode failures which are caused by faults occurring in a part of the system common to all redundant components.

2. Redundancy allocation

The use of redundancy does not immediately guarantee an improvement in the dependability of a system. The increase in weight, size and power consumption caused by redundancy can be quite severe. The increase in complexity may diminish the dependability improvement, unless a careful analysis is performed to show that a more dependable system can be obtained by allocating the redundant resources in a proper way.

A number of possibilities have to be examined to determine at which level redundancy needs to be provided and which components need to be made redundant. To understand the importance of these decisions, consider a serial system consisting of two components with reliabilities R_1 and R_2. If the system reliability $R = R_1R_2$ does not satisfy the design requirements, the designer may decide to make some of the components redundant. The possible choices of redundant configurations are shown in Figure 4.1(a) and (b). Assuming the component failures are mutually independent, the corresponding reliabilities of these systems are

$$R_a = (2R_1 - R_1^2)R_2$$
$$R_b = (2R_2 - R_2^2)R_1$$

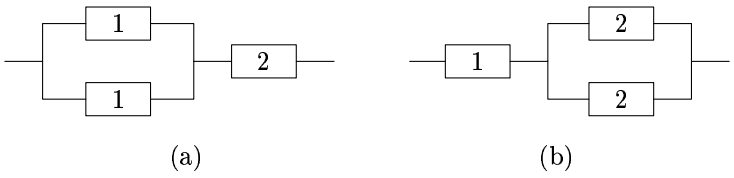

Figure 4.1. Redundancy allocation.

Taking the difference of R_b and R_a, we get

$$R_a - R_b = R_1 R_2 (R_2 - R_1)$$

It follows from this expression that the higher reliability is achieved if we duplicate the component that is least reliable. If $R_1 < R_2$, then configuration Figure 4.1(a) is preferable, and vice versa.

Another important parameter to examine is the *level of redundancy*. Consider the system consisting of three serial components. In high-level redundancy, the entire system in duplicated, as shown in Figure 4.2(a). In low-level redundancy, the duplication takes place at component level, as shown in Figure 4.2(b). If each of the block of the diagram is a subsystem, the redundancy can be placed at even lower levels.

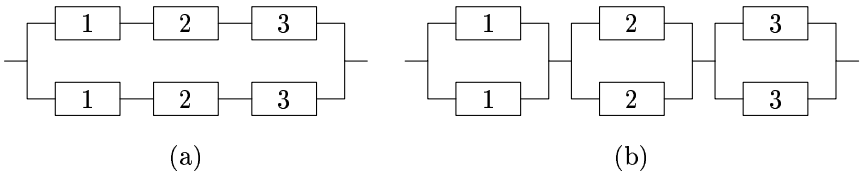

Figure 4.2. High-level and low-level redundancy.

Let us compare the reliabilities of the systems in Figures 4.2(a) and (b). Assuming that the component failures are mutually independent, we have

$$R_a = 1 - (1 - R_1 R_2 R_3)^2$$
$$R_b = (1 - (1 - R_1)^2)(1 - (1 - R_2)^2)(1 - (1 - R_3)^2)$$

The system in Figure 4.2(a) is a parallel combination of two serial sub-systems. The system in Figure 4.2(b) is a serial combination of three parallel sub-systems. As we can see, the reliabilities R_a and R_b differ, although the systems have the same number of components. If $R_A = R_B = R_C$, then the difference is

$$R_b - R_a = 6R^3(1 - R)^2$$

Consequently, $R_b > R_a$, i.e. low-level redundancy yields higher reliability than high-level redundancy. However, this dependency only holds if the components failures are truly independent in both configurations. In reality, common-mode failures are more likely to occur with low-level rather than with high-level redundancy, since in high-level redundancy the redundant units are normally isolated physically and therefore are less prone to common sources of stress.

3. Passive redundancy

Passive redundancy approach masks faults rather than detect them. Masking insures that only correct values are passed to the system output in spite of the

presence of a fault. In this section we first study the concept of triple modular redundancy, and then extend it to a more general case of N-modular redundancy.

3.1 Triple modular redundancy

The most common form of passive hardware redundancy is *triple modular redundancy* (TMR). The basic configuration is shown in Figure 4.3. The components are triplicated to perform the same computation in parallel. Majority voting is used to determine the correct result. If one of the modules fails, the majority voter will mask the fault by recognizing as correct the result of the remaining two fault-free modules. Depending on the application, the triplicated modules can be processors, memories, disk drives, buses, network connections, power supplies, etc.

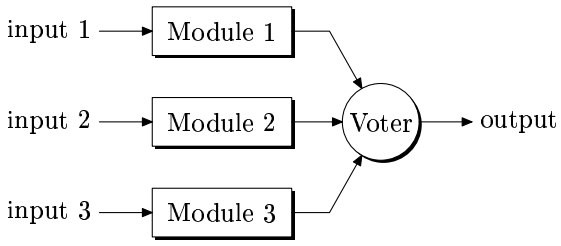

Figure 4.3. Triple modular redundancy.

A TMR system can mask only one module fault. A failure in either of the remaining modules would cause the voter to produce an erroneous result. In Section 5 we will show that the dependability of a TMR system can be improved by removing failed modules from the system.

TMR is usually used in applications where a substantial increase in reliability is required for a short period. For example, TMR is used in the logic section of launch vehicle digital computer (LVDC) of Saturn 5. Saturn 5 is a rocket carrying Apollo spacecrafts to the orbit. The functions of LVDC include the monitoring, testing and diagnosis of rocket systems to detect possible failures or unsafe conditions. As a result of using TMR, the reliability of the logic section for a 250-hr mission is approximately twenty times larger than the reliability of an equivalent simplex system. However, as we see in the next section, for longer duration missions, a TMR system is less reliable than a simplex system.

3.1.1 Reliability evaluation

The fact that a TMR system which can mask one module fault does not immediately imply that the reliability of a TMR system is higher than the reliability of a simplex system. To estimate the influences of TMR on reliability,

we need to take the reliability of modules as well as the duration of the mission into account.

A TMR system operates correctly as long as two modules operate correctly. Assuming that the voter is perfect and that the component failures are mutually independent, the reliability of a TMR systems is given by

$$R_{TMR} = R_1R_2R_3 + (1-R_1)R_2R_3 + R_1(1-R_2)R_3 + R_1R_2(1-R_3)$$

The term $R_1R_2R_3$ gives the probability that the first module functions correctly *and* the second module functions correctly *and* the third module functions correctly. The term $(1-R_1)R_2R_3$ stands for the probability that the first module has failed *and* the second module functions correctly *and* the third module functions correctly, etc. The overall probability is an *or* of the probabilities of the terms since the events are mutually exclusive. If $R_1 = R_2 = R_3 = R$, the above equation reduces to

$$R_{TMR} = 3R^2 - 2R^3 \tag{4.1}$$

Figure 4.4 compares the reliability of a TMR system R_{TMR} to the reliability of a simplex system consisting of a single module with reliability R. The

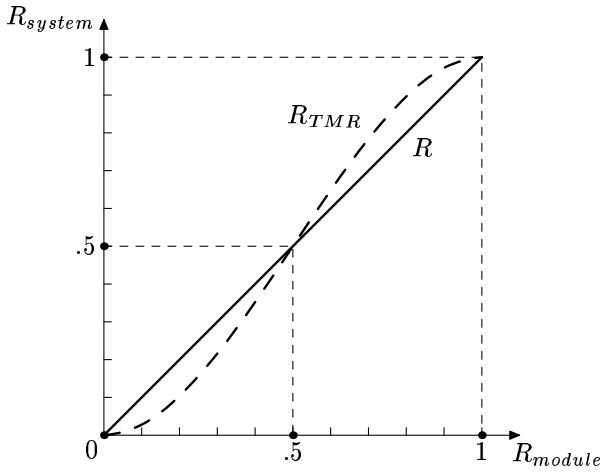

Figure 4.4. TMR reliability compared to simplex system reliability.

reliabilities of the modules composing the TMR system are assumed to be equal R. As can be seen, there is a point at which $R_{TMR} = R$. This point can be found by solving the equation $3R^2 - 2R^3 = R$. The three solutions are 0.5, 1 and 0, implying that the reliability of a TMR system is equal to the reliability of a simplex system when the reliability of the module is $R = 0.5$, when the module is perfect ($R = 1$), or when the module is failed ($R = 0$).

This further illustrates a difference between fault tolerance and reliability. A system can be fault-tolerant and still have a low overall reliability. For example, a TMR system build out of poor-quality modules with $R = 0.2$ will have a low reliability of $R_{TMR} = 0.136$. Vice versa, a system which cannot tolerate any faults can have a high reliability, e.g. when its components are highly reliable. However, such a system will fail as soon as the first fault occurs.

Next, let us consider how the reliability of a TMR system changes as a function of time. For a constant failure rate λ, the reliability of the system varies exponentially as a function of time $R(t) = e^{-\lambda t}$ (3.9). Substituting this expression in (4.1), we get

$$R_{TMR}(t) = 3e^{-2\lambda t} - 2e^{-3\lambda t} \tag{4.2}$$

Figure 4.5 shows how the reliabilities of simplex and TMR systems change as functions of time. The value of λt, rather than t is shown on the x-axis, to make

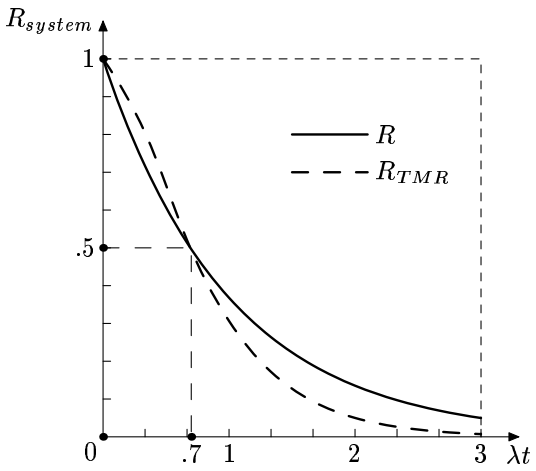

Figure 4.5. TMR reliability as a function of λt.

the comparison independent of the failure rate. Recall that $1/\lambda = MTTF$ (3.12), so that the point $\lambda t = 1$ corresponds to the time when the system is expected to experience the first failure. One can see that the reliability of the TMR system is higher than the reliability of the simplex system in the period between 0 and approximately $0.7\lambda t$. That is why TMR is suitable for applications whose mission time is shorter than 0.7 of MTTF.

3.1.2 Voting techniques

In the previous section we evaluated the reliability of a TMR system assuming that the voter is perfect. Clearly, such an assumption is not realistic. A more

precise estimation of the reliability of a TMR system takes the reliability of the voter into account:

$$R_{TMR} = (3R^2 - 2R^3)R_v$$

The voter is in series with the redundant modules, since if it fails, the whole system fails. The reliability of the voter must be very high in order to keep the overall reliability of the TMR system higher than the reliability of a corresponding simplex system. Fortunately, the voter is typically a very simple device compared to the redundant components and therefore its failure probability is much smaller. Still, in some systems the presence of a single point of failure is not acceptable by qualitative requirement specifications. We call *single point of failure* any component within a system whose failure leads to the failure of the system. In such cases, more complicated voting shemes are used. One possibility is to *decentralize* voting by having three voters instead of one, as shown in Figure 4.6. Decentralized voting avoids the single point of failure, but requires establishing consensus among three voters.

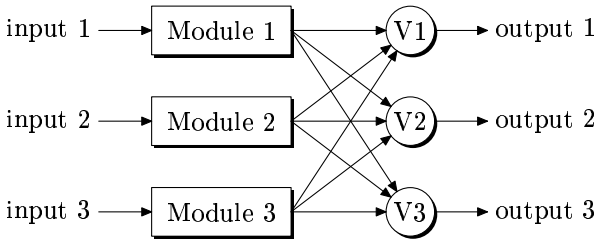

Figure 4.6. TMR system with three voters.

Another possibility is the so called *master-slave approach* that replaces a failed voter with a standby voter.

Voting heavily relies on an accurate timing. If values arrive at a voter at different times, incorrect voting result may be generated. Therefore, a reliable time service should be provided throughout a TMR or NMR system. This can be done either by using additional interval timers, or by implementing asynchronous protocols that rely on the progress of computation to provide an estimate of time. Multiple-processor systems should either provide a fault-tolerant global clock service that maintains a consistent source of time throughout the system, or to resolve time conflicts on an ad-hoc basis.

Another problem with voting is that the values that arrive at a voter may not completely agree, even in a fault-free case. For example, analog to digital converters may produce values which slightly disagree. A common approach to overcome this problem is to accept as correct the *median* value which lies between the remaining two. Another approach is to ignore several least significant bits of information and to perform voting only on the remaining bits.

Voting can be implemented in either hardware or software. Hardware voters are usually quick enough to meet any response deadline. If voting is done by software voters that must reach a consensus, adequate time may not be available. A hardware majority voter with 3 inputs for digital data is shown in Figure 4.7. The value of the output f is determined by the majority of the input values

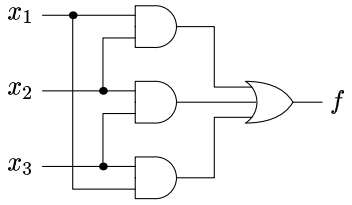

Figure 4.7. Logic diagram of a majority voter with 3 inputs.

x_1, x_2, x_3. The defining table for this voter is given in Table 4.1.

x_1	x_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Table 4.1. Defining table for 2-out-of-3 majority voter.

3.2 N-modular redundancy

N-modular redundancy (NMR) approach is based on the same principle as TMR, but uses n modules instead of three (Figure 4.8). The number n is usually selected to be odd, to make majority voting possible. A NMR system can mask $\lfloor N/2 \rfloor$ module faults.

Figure 4.9 plots the reliabilities of NMR systems for $n = 1, 3, 5$ and 7. Note that the x-axis shows the interval of time between 0 and $\lambda t = 1$, i.e. MTTF. This interval of time is of most interest for reliability analysis. As expected, larger values of n result in a higher increase of reliability of the system. At time approximately $0.7\lambda t$, the reliabilities of simplex, TMR, 5MR and 7MR system

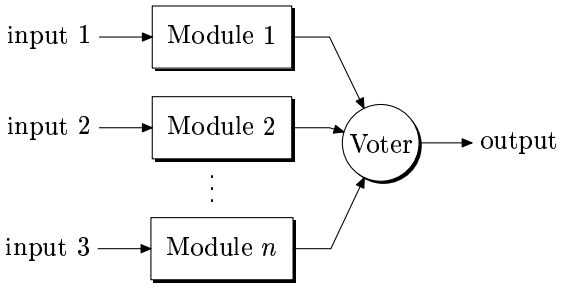

Figure 4.8. N-modular redundancy.

become equal. After $0.7\lambda t$, the reliability of a simplex system is higher than the reliabilities of redundant systems. So, similarly to TMR, NMR is suitable for applications with short mission times.

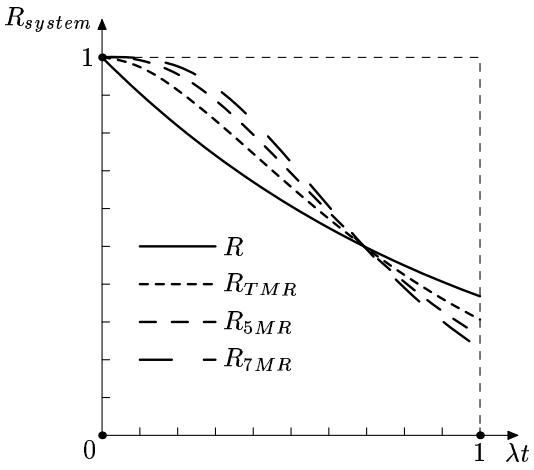

Figure 4.9. Reliability of an NMR system for different values of *n*.

4. Active redundancy

Active redundancy achieves fault tolerance by first detecting the faults which occur and then performing actions needed to recover the system back to the operational state. Active redundancy techniques are common in applications where temporary erroneous results are preferable to the high degree of redundancy required to achieve fault masking. Infrequent, occasional errors are allowed, as long as the system recovers back to normal operation in a specified interval of time.

In this section we consider three common active redundancy techniques: duplication with comparison, standby sparing and pair-and-a-spare, and examine the effect of redundancy on system dependability.

4.1 Duplication with comparison

The basic form of active redundancy is *duplication with comparison* shown in Figure 4.10. Two identical modules perform the same computation in parallel. The results of the computation are compared using a comparator. If the results disagree, an error signal is generated. Depending on the application, the duplicated modules can be processors, memories, I/O units, etc.

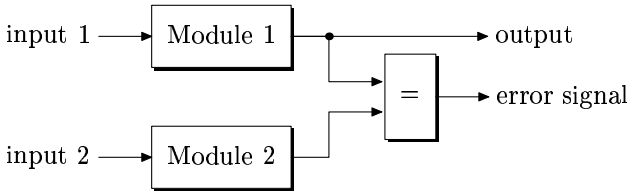

Figure 4.10. Duplication with comparison.

A duplication with comparison scheme can detect only one module fault. After the fault is detected, no actions are taken by the system to return back to the operational state.

4.1.1 Reliability evaluation

A duplication with comparison system functions correctly only until both modules operate correctly. When the first fault occurs, the comparator detects a disagreement and the normal functioning of the system stops, since the comparator is not capable to distinguish which of the results is the correct one. Assuming that the comparator is perfect and that the component failures are mutually independent, the reliability of the system is given by

$$R_{DC} = R_1 \cdot R_2 \tag{4.3}$$

or, $R_{DC} = R^2$ if $R_1 = R_2 = R$.

Figure 4.11 compares the reliability of a duplication with comparison system R_{DC} to the reliability of a simplex system consisting of a single module with reliability R. It can been seen that, unless the modules are perfect ($R(t) = 1$), the reliability of a duplication with comparison system is always smaller than the reliability of a simplex system.

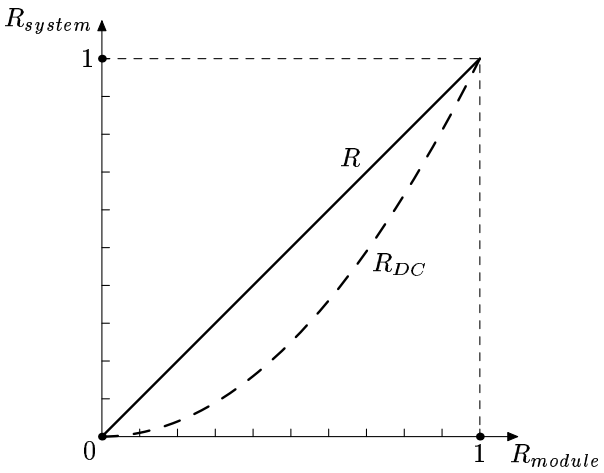

Figure 4.11. Duplication with comparison reliability compared to simplex system reliability.

4.2 Standby sparing

Standby sparing is another scheme for active hardware redundancy. The basic configuration is shown in Figure 4.12. Only one of *n* modules is operational and provides the system's output. The remaining $n-1$ modules serve as spares. A *spare* is a redundant component which is not needed for the normal system operation. A switch is a device that monitors the active module and switches operation to a spare if an error is reported by fault-detection unit FD.

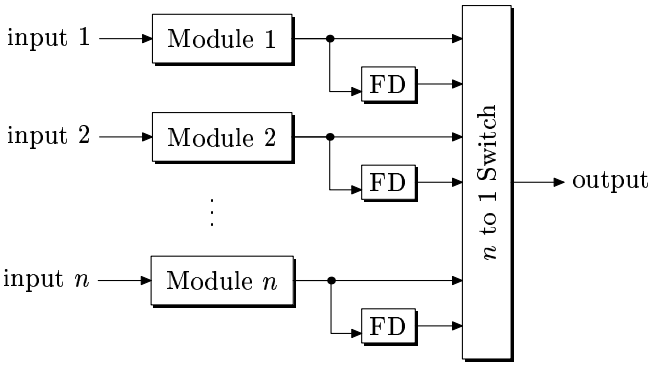

Figure 4.12. Standby sparing redundancy.

There are two types of standby sparing: hot standby and cold standby. In the *hot standby sparing*, both operational and spare modules are powered up.

The spares can be switched into use immediately after the operational module has failed. In the *cold standby sparing*, the spare modules are powered down until needed to replace the faulty module. A disadvantage of cold standby sparing is that time is needed to apply power to a module, perform initialization and re-computation. An advantage is that the stand-by spares do not consume power. This is important in applications like satellite systems, where power consumption is critical. Hot standby sparing is preferable where the momentary interruption of normal operation due to reconfiguration needs to be minimized, like in a nuclear plant control system.

A standby sparing system with n modules can tolerate $n-1$ module faults. Here by "tolerate" we mean that the system will detect and locate the faults, successfully recover from them and continue delivering the correct service. When the nth fault occurs, it will still be detected, but the system will not be able to recover back to normal operation.

The standby sparing redundancy technique is used in many systems. One example is the Apollo spacecraft's telescope mount pointing computer. In this system, two identical computers, an active and a spare, are connected to a switching device that monitors the active computer and switches operation to the backup in case of a malfunction.

Another example of using standby sparing is Saturn 5 launch vehicle digital computer (LVDC) memory section. The section consists of two memory blocks, with each memory being controlled by an independent buffer register and parity-checked. Initially, only one buffer register output is used. When a parity error is detected in the memory being used, operation immediately transfers to the other memory. Both memories are then re-generated by the buffer register of the "correct" memory, thus correcting possible transient faults.

Standby sparing is also used in Compaq's NonStop Himalaya server. The system is composed of a cluster of processors working in parallel. Each processor has its own memory and copy of the operating system. A primary process and a backup process are run on separate processors. The backup process mirrors all the information in the primary process and is able to instantly take over in case of a primary processor failure.

4.2.1 Reliability evaluation

By their nature, standby systems involve dependency between components, since the spare units are held in reserve and only brought to operation in the event the primary unit fails. Therefore, standby systems are best analyzed using Markov models. We first consider an idealized case when the switching mechanism is perfect. We also assume that the spare cannot fail while it is in the standby mode. Later, we consider the possibility of failure during switching.

Perfect switching case

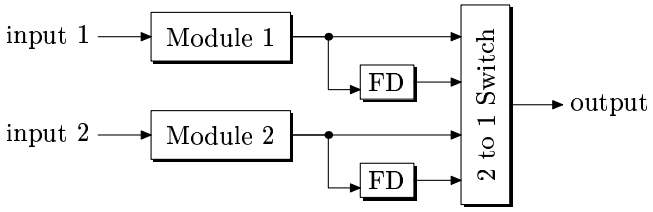

Figure 4.13. Standby sparing system with one spare.

Consider a standby sparing scheme with one spare, shown in Figure 4.13. Let module 1 be a primary module and module 2 be a spare. The state transition diagram of the system is shown in Figure 4.14. The states are numbered according to the Table 4.2.

Component		State
1	2	Number
O	O	1
F	O	2
F	F	3

Table 4.2. Markov states of the state transition diagram of a standby sparing system with one spare.

When the primary component fails, there is a transition between state 1 and state 2. If a system is in state 2 and the spare fails, there is a transition to state 3. Since we assumed that the spare cannot fail while in standby mode, the combination (O,F) cannot occur. The states 1 and 2 are operational states. The state 3 is the failed state.

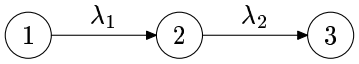

Figure 4.14. State transition diagram of a standby sparing system with one spare.

The transition matrix for the state transition diagram 4.14 is given by

$$\mathbf{M} = \begin{bmatrix} -\lambda_1 & 0 & 0 \\ \lambda_1 & -\lambda_2 & 0 \\ 0 & \lambda_2 & 0 \end{bmatrix}.$$

So, we get the following system of state transition equations

$$\frac{d}{dt}\begin{bmatrix} P_1(t) \\ P_2(t) \\ P_3(t) \end{bmatrix} = \begin{bmatrix} -\lambda_1 & 0 & 0 \\ \lambda_1 & -\lambda_2 & 0 \\ 0 & \lambda_2 & 0 \end{bmatrix} \cdot \begin{bmatrix} P_1(t) \\ P_2(t) \\ P_3(t) \end{bmatrix}$$

or

$$\begin{cases} \frac{d}{dt}P_1(t) = -\lambda_1 P_1(t) \\ \frac{d}{dt}P_2(t) = \lambda_1 P_1(t) - \lambda_2 P_2(t) \\ \frac{d}{dt}P_3(t) = \lambda_2 P_2(t) \end{cases}$$

By solving the system of equations, we get

$$\begin{aligned} P_1(t) &= e^{-\lambda_1 t} \\ P_2(t) &= \tfrac{\lambda_1}{\lambda_2-\lambda_1}(e^{-\lambda_1 t} - e^{-\lambda_2 t}) \\ P_3(t) &= 1 - \tfrac{1}{\lambda_2-\lambda_1}(\lambda_2 e^{-\lambda_1 t} - \lambda_1 e^{-\lambda_2 t}) \end{aligned}$$

Since $P_3(t)$ is the only state corresponding to system failure, the reliability of the system is the sum of $P_1(t)$ and $P_2(t)$

$$R_{SS}(t) = e^{-\lambda_1 t} + \frac{\lambda_1}{\lambda_2 - \lambda_1}(e^{-\lambda_1 t} - e^{-\lambda_2 t})$$

This can be re-written as

$$R_{SS}(t) = e^{-\lambda_1 t} + \frac{\lambda_1}{\lambda_2 - \lambda_1} e^{-\lambda_1 t}(1 - e^{-(\lambda_2-\lambda_1)t}) \tag{4.4}$$

Assuming $(\lambda_2 - \lambda_1)t << 1$, we can expand the term $e^{-(\lambda_2-\lambda_1)t}$ as a power series of $-(\lambda_2 - \lambda_1)t$ as

$$e^{-(\lambda_2-\lambda_1)t} = 1 - (\lambda_2 - \lambda_1)t + 1/2(\lambda_2 - \lambda_1)^2 t^2 - \ldots$$

Substituting it in (4.4), we get

$$R_{SS}(t) = e^{-\lambda_1 t} + \lambda_1 e^{-\lambda_1 t}(t - 1/2(\lambda_1 - \lambda_2)t^2 + \ldots).$$

Assuming $\lambda_2 = \lambda_1$, the above can be simplified to

$$R_{SS}(t) = (1 + \lambda t)e^{-\lambda t} \tag{4.5}$$

Next, let us see how the equation (4.5) would change if we would ignore the dependency between the failures. If the primary and spare module failures are treated as mutually independent, the reliability of a standby sparing system is a sum of two probabilities:

1 The probability that module 1 operates correctly, and

2 The probability that module 2 operates correctly, while module 1 has failed and has been replaced by module 2.

Then, we get the following expression:

$$R_{SS} = R_1 + (1 - R_1)R_2$$

If $R_1 = R_2 = R$, then

$$R_{SS} = 2R - R^2$$

or

$$R_{SS}(t) = 2e^{-\lambda t} - e^{-2\lambda t} \tag{4.6}$$

Figure 4.15 compares the plots of the reliabilities (4.5) and (4.6). One can see that neglecting the dependencies between failures leads to underestimating the standby sparing system reliability.

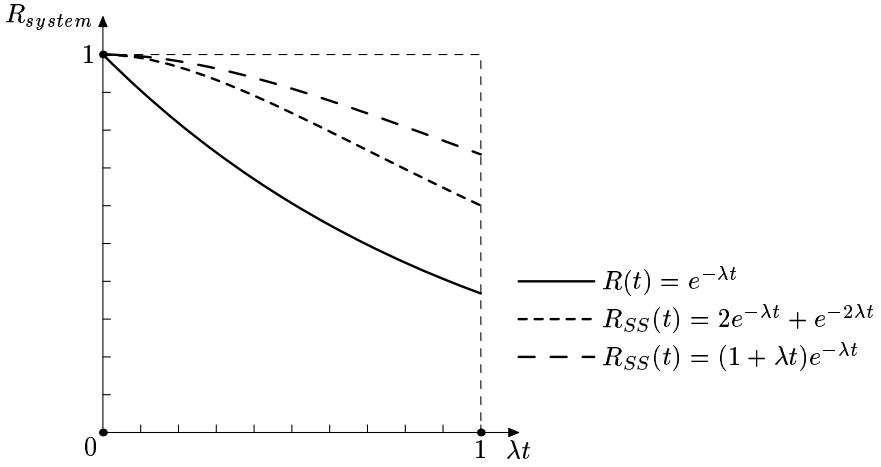

Figure 4.15. Standby sparing reliability compared to simplex system reliability.

Non-perfect switching case

Next, we consider the case when the switch is not perfect. Suppose that the probability that the switch successfully replaces the primary unit by a spare is p. Then, the probability that the switch fails to do it is $1 - p$. The state transition diagram with these assumptions is shown in Figure 4.16. The transition from state 1 is partitioned into two transitions. The failure rate is multiplied by p to get the rate of successful transition to state 2. The failure rate is multiplied by $1 - p$ to get the rate of the switch failure.

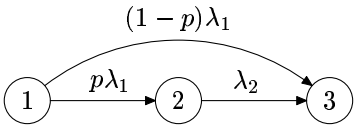

Figure 4.16. State transition diagram of a standby system with one spare.

The state transition equations corresponding to the state transition diagram (4.16) are

$$\begin{cases} \frac{d}{dt}P_1(t) = -\lambda_1 P_1(t) \\ \frac{d}{dt}P_2(t) = p\lambda_1 P_1(t) - \lambda_2 P_2(t) \\ \frac{d}{dt}P_3(t) = \lambda_2 P_2(t) + (1-p)\lambda_1 P_1(t) \end{cases}$$

By solving this system of equations, we get

$$\begin{aligned} P_1(t) &= e^{-\lambda_1 t} \\ P_2(t) &= \tfrac{p\lambda_1}{\lambda_2-\lambda_1}(e^{-\lambda_1 t} - e^{-\lambda_2 t}) \\ P_3(t) &= 1 - (1 + \tfrac{p\lambda_1}{\lambda_2-\lambda_1})e^{-\lambda_1 t} + \tfrac{p\lambda_1}{\lambda_2-\lambda_1}e^{-\lambda_2 t} \end{aligned}$$

As before, $P_3(t)$ corresponds to system failure. So, the reliability of the system is the sum of $P_1(t)$ and $P_2(t)$

$$R_{SS}(t) = e^{-\lambda_1 t} + \frac{p\lambda_1}{\lambda_2 - \lambda_1}(e^{-\lambda_1 t} - e^{-\lambda_2 t})$$

Assuming $\lambda_2 = \lambda_1$, the above can be simplified to

$$R_{SS}(t) = (1 + p\lambda t)e^{-\lambda t} \tag{4.7}$$

Figure 4.17 compares the reliability of a standby sparing system for different values of p. As p decreases, the reliability of the standby sparing system decreases. When p reaches zero, the standby sparing system reliability reduces to the reliability of a simplex system.

4.3 Pair-and-a-spare

Pair-and-a-spare technique combines standby sparing and duplication and comparison approaches (Figure 4.18). The idea is similar to standby sparing, however two modules instead of one are operated in parallel. As in the duplication with comparison case, the results are compared to detect disagreement. If an error signal is received from the comparator, the switch analyzes the report

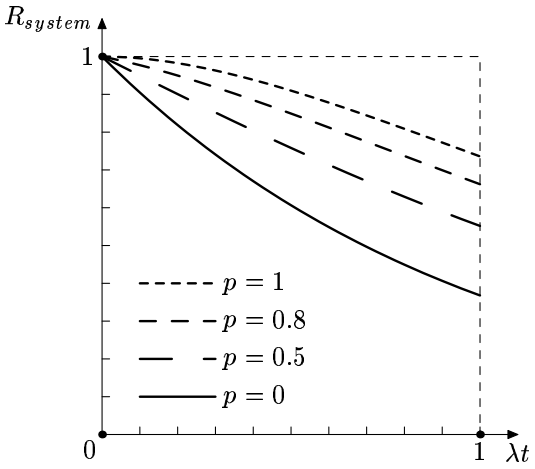

Figure 4.17. Reliability of a standby sparing system for different values of p.

from the fault detection block and decides which of the two modules' output is faulty. The faulty module is removed from operation and replaced with a spare module.

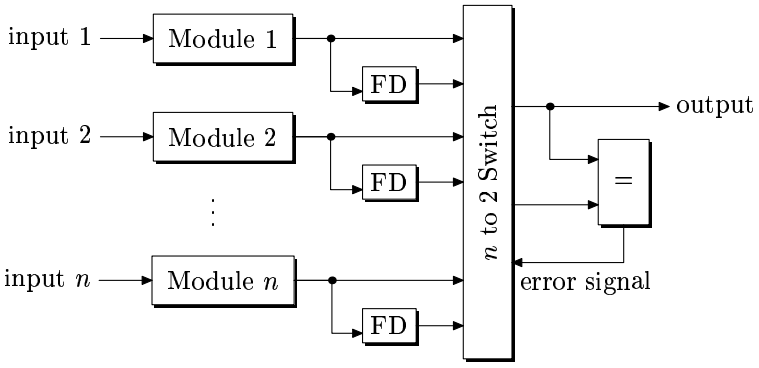

Figure 4.18. Pair-and-a-spare redundancy.

A pair-and-a-spare system with n modules can tolerate $n-1$ module faults. When the $n-1$th fault occurs, it will be detected and located by the switch and the correct result will be passed to the system's output. However, since there will be no more spares available, the switch will not be able to replace the faulty module with a spare module. The system's configuration will be reduced to a simplex system with one module. So, the nth fault will not be detected.

5. Hybrid redundancy

The main idea of hybrid redundancy is to combine the attractive features of passive and active approach. Fault masking is used to prevent system from producing momentary erroneous results. Fault detection, location and recovery are used to reconfigure the system after a fault occurs. In this section, we consider three basic techniques for hybrid redundancy: self-purging redundancy, N-modular redundancy with spares and triplex-duplex redundancy.

5.1 Self-purging redundancy

Self-purging redundancy consists of n identical modules which are actively participating in voting (Figure 4.19). The output of the voter is compared to the outputs of individual modules to detect disagreement. If a disagreement occurs, the switch opens and removes, or *purges*, the faulty module from the system. The voter is designed as a threshold gate, capable to adapt to the changing number of inputs. The input of the removed module is forced to zero and therefore do not contribute to the voting.

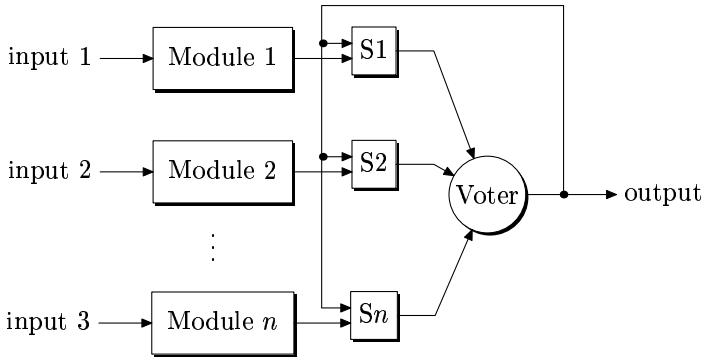

Figure 4.19. Self-purging redundancy.

A self-purging redundancy system with n modules can mask $n-2$ module faults. When $n-2$ modules are purged and only two are left, the system will be able to detect the next, $n-1$th fault, but, as in the duplication with comparison case, the voter will not be able to distinguish which one of the two results is the correct one.

5.1.1 Reliability evaluation

Since all the modules of the system operate in parallel, we can assume that the modules' failures are mutually independent. It is sufficient that two of the modules of the system function correctly for the system to be operational. If the voter and the switches are perfect, and if all the modules have the same

reliability $R_1 = R_2 = \ldots = R_n = R$, then the system is *not* reliable if all the modules have failed (probability $(1-R)^n$), or if all but one modules have failed (probability $R(1-R)^{n-1}$). Since there are n choices for one of n modules to remain operational, we get the equation

$$R_{SP} = 1 - ((1-R)^n + nR(1-R)^{n-1}) \tag{4.8}$$

Figure 4.20 compares the reliabilities of self-purging redundancy systems with three, five and seven modules.

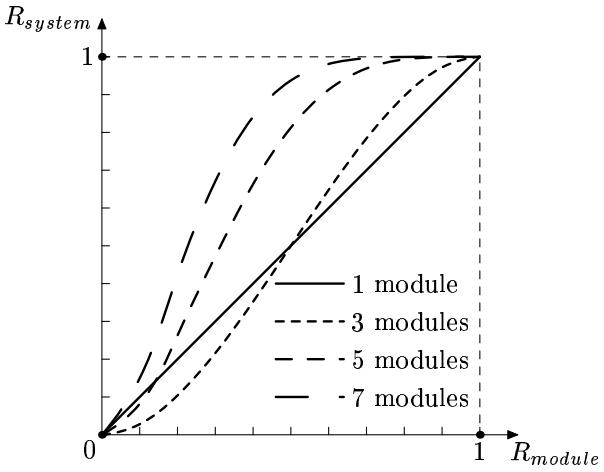

Figure 4.20. Reliability of a self-purging redundancy system with 3, 5 and 7 modules.

5.2 N-modular redundancy with spares

N-modular redundancy with k spares is similar to self-purging redundancy with $k+n$ modules, except that only n modules provide input to a majority voter (Figure 4.21). Additional k modules serve as spares. If one of the primary modules becomes faulty, the voter will mask the erroneous result and the switch will replace the faulty module with a spare one. Various techniques are used to identify faulty modules. One approach is to compare the output of the voter with the individual outputs of the modules, as shown in Figure 4.21. A module which disagrees with the majority is declared faulty.

The fault-tolerant capabilities of an N-modular redundancy system with k spares depend on the form of voting used as well as the implementation of the switch and comparator. One possibility is that, after the spares are exhausted, the disagreement detector is switched off and the system continues working as a passive NMR system. Then, such a system can mask $\lfloor n/2 \rfloor + k$ faults,

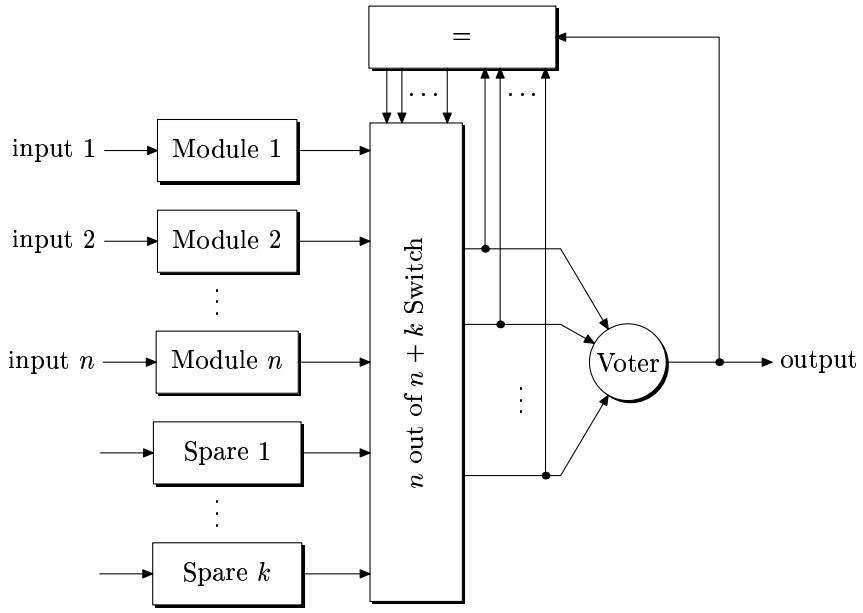

Figure 4.21. N-modular redundancy with spares.

i.e. the number of faults a NMR system can mask plus the number of spares. Another possibility is that the disagreement detector remains on, but the voter is designed to be capable to adjust to the decreasing number of inputs. In this case, the behavior of the system is similar to the behavior of a self-purging system with $n+k$ modules, i.e. up to $k+n-2$ module faults can be masked. Suppose the spares are exhausted after the first k faults, and the $k+1$th fault occurred. As before, the erroneous result will be masked by the voter, the output of the voter will be compared to the individual outputs of the modules, and the faulty will be removed from considerations. A difference is that it will not be replaced with a spare one, but instead the system will continue working as $n-1$-modular system. Then a $k+i$th fault occurs, the voter votes on $n-i$) modules.

5.3 Triplex-duplex redundancy

Triplex-duplex redundancy combines triple modular redundancy and duplication with comparison (Figure 4.22). A total of six identical modules, grouped in three pairs, are computing in parallel. In each pair, the results of the computation are compared using a comparator. If the results agree, the output of the comparator participates in the voting. Otherwise, the pair of modules is declared faulty and the switch removes the pair from the system. In this way, only faulty-free pair participates in voting.

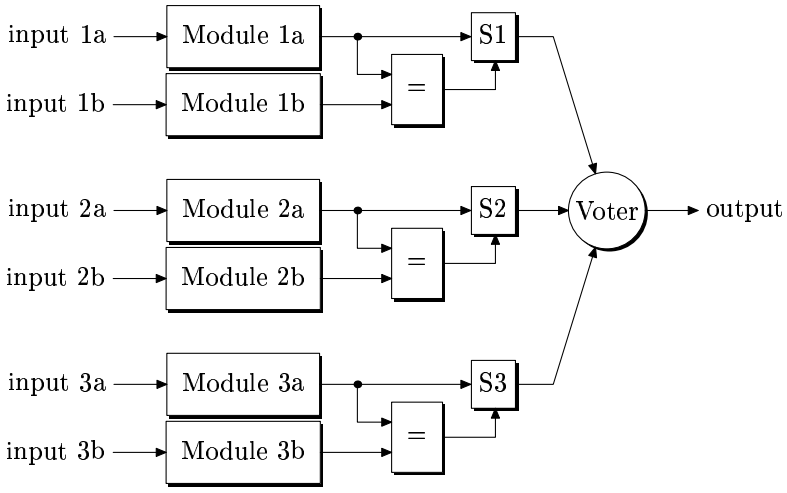

Figure 4.22. Triplex-duplex redundancy.

6. Problems

4.1. Explain the difference between passive, active and hybrid hardware redundancy. Discuss the advantages and disadvantages of each approach.

4.2. Suppose that in the system shown in Figure 4.1 the two components have the same cost and $R_1 = 0.75$, $R_2 = 0.96$. If it is permissible to add two components to the system, would it be preferable to replace component 1 by a three-component parallel system, or to replace components 1 and 2 each by two-component parallel systems?

4.3. A disk drive has a constant failure rate and an MTTF of 5500 hr.

(a) What is the probability of failure for one year of operation?

(b) What is the probability of failure for one year of operation if two of the drives are placed in parallel and the failures are independent?

4.4. Construct the Markov model of the TMR system with three voters shown in Figure 4.6. Assume that the components are independent. The failure rate of the modules is λ_m. The failure rates of the voters is λ_v. Derive and solve the system of state transition equations representing this system. Compute the reliability of the system.

4.5. Draw a logic diagram of a majority voter with 5 inputs.

4.6. Suppose the design life reliability of a standby system consisting of two identical units should be at least 0.97. If the MTTF of each unit is 6 months,

determine the design life time. Assume that the failures are independent and ignore switching failures.

4.7. An engineer designs a system consisting of two subsystems in series with the reliabilities $R_1 = 0.99$ and $R_2 = 0.85$. The cost of the two subsystems is approximately the same. The engineer decides to add two redundant components. Which of the following is the best to do:

(a) Duplicate subsystems 1 and 2 in high-level redundancy (Figure 4.2(a)).

(b) Duplicate subsystems 1 and 2 in low-level redundancy (Figure 4.2(b)).

(c) Replace the second subsystem by a three-component parallel system.

4.8. A computer with MTTF of 3000 hr is to operate continuously on a 500 hr mission.

(a) Compute computer's mission reliability.

(b) Suppose two such computers are connected in a standby configuration. If there are no switching failures and no failures of the backup computer while in the standby mode, what is the system MTTF and the mission reliability?

(c) What is the mission reliability if the probability of switching failure is 0.02?

4.9. A chemical process control system has a reliability of 0.97. Because reliability is considered too low, a redundant system of the same design is to be installed. The design engineer should choose between a parallel and a standby configuration. How small must the probability of switching failure be for the standby configuration to be more reliable than the parallel configuration? Assume that there is no failures of the backup system while in the standby mode.

4.10. Give examples of applications where you would recommend to use cold standby sparing and hot standby sparing (two examples each). Justify your answer.

4.11. Compare the MTTF of a standby spare system with 3 modules and a pair-and-a-spare system with 3 modules, provided the failure rate of a single module is 0.01 failures per hour. Assume the modules obey the exponential failure law. Ignore the switching failures and the dependency between the module's failures.

4.12. A basic non-redundant controller for a heart pacemaker consists of an analog to digital (A/D) converter, a microprocessor and a digital to analog (D/A) converter. Develop a design making the controller tolerant to any two component faults (component here means A/D converter, microprocessor or D/A

converter). Show the block diagram of your design and explain why you recommend it.

4.13. Construct the Markov model of a hybrid N-modular redundancy with 3 active modules and one spare. Assume that the components are independent and that the probability that the switch successfully replaces the failed module by a spare is p.

4.14. How many faulty modules can you tolerate in:

(a) 5-modular passive redundancy?

(b) standby sparing redundancy with 5 modules?

(c) self-purging hybrid modular redundancy with 5 modules?

4.15. Design a switch for hybrid N-modular redundancy with 3 active modules and 1 spare.

4.16. (a) Draw a diagram of standby sparing active hardware redundancy technique with 2 spares.

(b) Using Markov models, write an expression for the reliability of the system you showed on the diagram for

(a) perfect switching case,

(b) non-perfect switching case.

(c) Calculate the reliabilities for (a) and (b) after 1000 hrs for the failure rate $\lambda = 0.01$ per 100 hours.

4.17. Which redundancy would you recommend to combine with self-purging hybrid hardware redundancy to distinguish between transient and permanent faults? Briefly describe what would be the main benefit of such a combination.

4.18. Calculate the MTTF of a 5-modular hardware redundancy system, provided the failure rate of a single module is 0.001 failures per hour. Assume the modules obey the exponential failure law. Compare the MTTF of the 5-modular redundancy system with the MTTF of a 3-modular hardware redundancy system having the failure rate 0.01 failures per hour.

4.19. Draw a simplified Markov model for the 5-modular hardware redundancy scheme with failure rate λ. Explain which state of the system each of the nodes in your chain represents.

Chapter 5

INFORMATION REDUNDANCY

The major difference between a thing that might go wrong and a thing that cannot possibly go wrong is that when a thing that cannot possibly go wrong goes wrong it usually turns out to be impossible to get at or repair.

—Douglas Adams, "Mostly Harmless"

1. Introduction

In this chapter we study how fault-tolerance can be achieved by means of encoding. Encoding is powerful technique which helps us to avoid unwanted information changes during storage or transmission. Attaching special check bits to blocks of digital information enables special-purpose hardware to detect and correct a number of communication and storage faults, such as changes in single bits or changes to several adjacent bits. Parity code used for random access memories in computer systems is a common example of an application of encoding. Other examples are communication protocols that provide a variety of detection and correction options including the encoding of large blocks of data to withstand multiple faults and provisions for multiple retries in the case the error correcting facilities cannot cope with the faults.

Coding theory was originated in the late 1940s, by two seminal works by Hamming and Shannon. Hamming, working at Bell Laboratories in the USA, was studying possibilities for protecting storage devices from the corruption of a small number of bits by a code which would be more efficient than simple repetition. He realized the need to consider sets of words, or *codewords*, where every pair differs in a large number of bit positions. Hamming defined the notion of distance between two words and observed this was a metric, thus leading to interesting properties. This distance is now called *Hamming distance*. His first attempt produced a code in which four data bits were followed by three check

bits which allowed not only the detection but the correction of a single error. The repetition code would require nine check bits to achieve this. Hamming published his results in 1950.

Slightly prior to Hamming's publication, in 1948, Shannon, also at Bell Labs, wrote an article formulating the mathematics behind the theory of communication. In this article, he developed probability and statistics to formalize the notion of information. Then, he applied this notion to study how a sender can communicate efficiently over different media, or more generally, channels of communication to a receiver. The channels under consideration were of two different types: noiseless or noisy. In the former case, the goal is to compress the information at the sender's end and to minimize the total number of symbols communicated while allowing the receiver to recover transmitted information correctly. The later case, which is more important to the topic of this book, considers a channel that alters the signal being sent by adding to it a *noise*. The goal in this case is to add some redundancy to the message being sent so that a few erroneous symbols at the receiver's end still allow the receiver to recover the sender's intended message. Shannon's work showed, somewhat surprisingly, that the same underlying notions captured the *rate* at which one could communicate over either class of channels. Shannon's methods involved encoding messages using long random strings, and the theory relied on the fact that long messages chosen at random tend to be far away from each other. Shannon had shown that it was possible to encode messages in such a way that the number of extra bits transmitted was as small as possible.

Although Shannon's and Hamming's works were chronologically and technically interwined, both researchers seem to regard the other as far away from their own work. Shannon's papers never explicitly refers to distance in his main technical results. Hamming, in turn, does not mention the applicability of his results to reliable computing. Both works, however, were immediately seen to be of monumental impact. Shannon's results started driving the theory of communication and storage of information. This, in turn, became the primary motivation for much research in the theory of error-correcting codes.

The value of error-correcting codes for transmitting information became immediately apparent. A wide variety of codes were constructed, achieving both economy of transmission and error-correction capacity. Between 1969 and 1973 the NASA Mariner probes used a powerful Reed-Muller code capable of correcting 7 errors out of 32 bits transmitted. The codewords consisted of 6 data bits and 26 check bits. The data was sent to Earth at the rate over 16,000 bits per second.

Another application of error-correcting codes came with the development of the compact disk (CD). To guard against scratches, cracks and similar damage two "interleaved" codes which can correct up to 4,000 consecutive errors (about 2.5 mm of track) are used.

Code selection is usually guided by the types of errors required to be tolerated and the overhead associated with each of the error detection techniques. For example, error correction is a common level of protection for minicomputers and mainframes whereas the cheaper error detection by parity code is more common in microcomputers. For solid state disks, storing system's critical, non-recoverable files, the most popular codes are Hamming codes to correct errors in main memory, and Reed-Solomon codes to correct errors in peripheral devices such as tape and disk storage.

2. Fundamental notions

In this section, we introduce the basic notions of coding theory. We assume that our data is in the form of strings of binary bits, 0 or 1. We also assume that the errors occur randomly and independently from each other, but at a predictable overall rate.

2.1 Code

A *binary code of length n* is a set of binary *n*-tuples satisfying some well-defined set of rules. For example, an even parity code contains all *n*-tuples that have an even number of 1s. The set $B^n = \{0,1\}^n$ of all possible 2^n binary *n*-tuples is called *codespace*.

A *codeword* is an element of the codespace satisfying the rules of the code. To make error-detection and error-correction possible, codewords are chosen to be a nonempty subset of all possible 2^n binary *n*-tuples. For example, a parity code of length *n* has 2^{n-1} codewords, which is one half of all possible 2^n *n*-tuples. An *n*-tuple not satisfying the rules of the code is called a *word*.

The number of codewords in a code *C* is called the *size* of C, denoted by $|C|$.

2.2 Encoding

Encoding is the process of computing a codeword for a given data. An encoder takes a binary *k*-tuple representing the data and converts it to a codeword using the rules of the code. For example, to compute a codeword for an even parity code, the parity of the data is first determined. If the parity is odd, a 1-bit is attached to the end of the *k*-tuple. Otherwise, a 0-bit is attached. The difference $n-k$ between the length *n* of the codeword and the length *k* of the data gives the number of *check bits* which must be added to the data to do the encoding. *Separable code* is the code in which the check bits can be clearly separated from the data bits. Parity code is an example of a separable code. *Non-separable code* is a code in which the check bits cannot be separated from the data bits. Cyclic code is an example of a non-separable code.

2.3 Information rate

To encode binary k-bit data, we need a code consisting of at least 2^k codewords, since any data word should be assigned its own individual codeword from C. Vice versa, a code of size $|C|$ encodes the data of length $k \leq \lceil \log_2 |C| \rceil$ bits. The ratio k/n is called the *information rate* of the code. The information rate determines the redundancy of the code. For example, a repetition code obtained by repeating the data three times, has the information rate $1/3$. Only one out of three bits carries the message, the other two are redundant.

2.4 Decoding

Decoding is the process of restoring data encoded in a given codeword. A decoder reads a codeword and recovers the original data using the rules of the code. For example, a decoder for a parity code truncates the codeword by one bit.

Suppose that an error has occurred and a non-codeword is received by a decoder. A usual assumption in coding theory is that a pattern of errors that involves a small number of bits is more likely to occur than any pattern that involves a large number of bits. Therefore, to perform decoding, we search for a codeword which is "closest" to the received word. Such a technique is called *maximum likelihood decoding*. As a measure of distance between two binary n-tuples x and y we use the Hamming distance.

2.5 Hamming distance

The *Hamming distance* between two binary n-tuples, x and y, denoted by $\delta(x,y)$, is the number of bit positions in which the n-tuples differ. For example, $x = 0011$ and $y = 0101$ differ in 2 bit positions, so $\delta(x,y) = 2$. Hamming distance gives us an estimate of how many bit errors have to occur to change x into y.

Hamming distance is a genuine metric on the codespace B^n. A *metric* is a function that associates any two objects in a set with a number and that preserves a number of properties of the distance with which we are familiar. These properties are formulated in the following three axioms:

1 $\delta(x,y) = 0$ if and only if $x = y$.

2 $\delta(x,y) = \delta(y,x)$.

3 $\delta(x,y) + \delta(y,z) \geq \delta(x,z)$.

The metric properties of the Hamming distance allow us to use the geometry of the codespace to reason about the codes. As an example, consider the codespace B^3 presented by a three-dimensional cube shown in Figure 5.1. Codewords $\{000, 011, 101, 110\}$ are marked with large solid dots. Adjacent

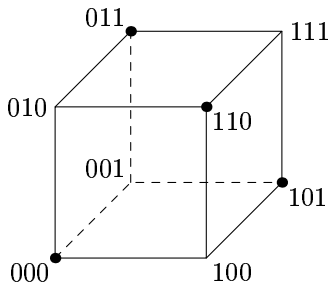

Figure 5.1. Code $\{000, 011, 101, 110\}$ in the codespace B^3.

vertices differ by a single bit. It is easy to see that the Hamming distance satisfies the metric properties listed above, e.g. $\delta(000,011)+\delta(011,111) = 2+1 = 3 = \delta(000,111)$.

2.6 Code distance

The *code distance* of a code C is the minimum Hamming distance between any two distinct pairs of codewords of C. For example, the code distance of a parity code equals two. The code distance determines the error detecting and error correcting capabilities of a code. For instance, consider the code $\{000, 011, 101, 110\}$ in Figure 5.1. The code distance of this code is two. Any one-bit error in any codeword produces a word laying on distance one from the affected codeword. Since all codewords are on distance two from each other, the error will be detected.

As another example, consider the code $\{000, 111\}$ shown in Figure 5.2. The

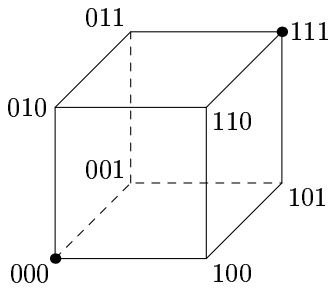

Figure 5.2. Code $\{000, 111\}$ in the codespace B^3.

codewords are marked with large solid dots. Suppose an error occurred in the first bit of the codeword 000. The resulting word 100 is on distance one from

000 and on distance two from 111. Thus, we correct 100 to the codeword 000, which is closest to 100 according to the Hamming distance.

The code $\{000, 111\}$ is a replication code, obtained by repeating the data three times. Only one of the bits of a codeword carries the data, the other two are redundant. By its nature, this redundancy is similar to TMR, but it is implemented in the information domain. In TMR, the voter compares the output values of the modules. In a replication code, a decoder analyzes the bits of the received word. In both cases, the majority of values of bits determines the decision.

In general, to be able to correct ε-bit errors, a code should have the code distance of at least $2\varepsilon + 1$. To be able to detect ε-bit errors, the code distance should be at least $\varepsilon + 1$.

2.7 Code efficiency

Throughout the chapter, we evaluate the efficiency of a code using the following three criteria:

1 Number bit errors a code can detect/correct, reflecting the fault tolerant capabilities of the code.

2 Information rate k/n, reflecting the amount of information redundancy added.

3 Complexity of encoding and decoding schemes, reflecting the amount of hardware, software and time redundancy added.

The first item in the list above is the most important. Ideally, we would like to have a code that is capable of correcting all errors. The second objective is an efficiency issue. We would rather not waste resources by exchanging data on a very low rate. Easy encoding and decoding schemes are likely to have a simple implementation in either hardware or software. They are also desirable for efficiency reasons. In general, the more errors that a code needs to correct per message digit, the less efficient the communication and usually the more complicated the encoding and decoding schemes. A good code balances these objectives.

3. Parity codes

Parity codes are the oldest family of codes. They have been used to detect errors in the calculations of the relay-based computers in late 1940's.

The *even (odd) parity code* of length n is composed of all the binary n-tuples that contain an even (odd) number of 1's. Any subset of $n-1$ bits of a codeword can be viewed as data bits, carrying the information, while the remaining nth bit checks the parity of the codeword. Any single-bit error can be detected, since the parity of the affected n-tuple will be odd (even) rather than even (odd). It is

not possible to locate the position of the erroneous bit. Thus, it is not possible to correct it.

The most common application of parity is error-detection in memories of computer systems. A diagram of a memory protected by a parity code is shown in Figure 5.3.

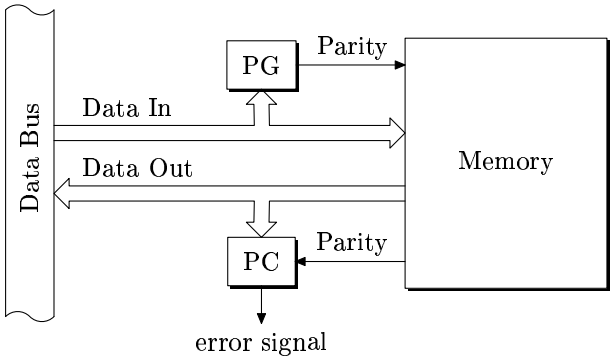

Figure 5.3. A memory protected by a parity code; PG = parity generator; PC = parity checker.

Before being written into a memory, the data is encoded by computing its parity. In most computer systems, one parity bit per byte (8 bits) of data is computed. The generation of parity bits is done by a parity generator (PG) implemented as a tree of exclusive-OR (XOR) gates. Figure 5.4 shows a logic diagram of an even parity generator for 4-bit data (d_0, d_1, d_2, d_3).

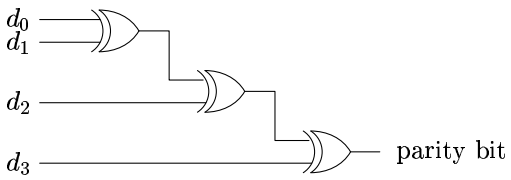

Figure 5.4. Logic diagram of a parity generator for 4-bit data (d_0, d_1, d_2, d_3).

When data is written into memory, parity bits are written along with the corresponding bytes of data. For example, for a 32-bit word size, four parity bits are attached to data and a 36-bit codeword is stored in the memory. Some systems, like Pentium processor, have a 64-bit wide memory data path. In these case, eight parity bits are attached to data. The resulting codeword 72 bit long.

When the data is read back from the memory, parity bits are re-computed and the result is compared to the previously stored parity bits. Re-computation of parity is done by a parity checker (PC). Figure 5.5 shows a logic diagram

of an even parity checker for 4-bit data (d_0, d_1, d_2, d_3). The logic diagram is similar to the one of a parity generator, except that one more XOR gate is added to compare the re-computed parity bit to the previously stored parity bit. If the parity bits disagree, the output of the XOR gate is 1. Otherwise, the output is 0.

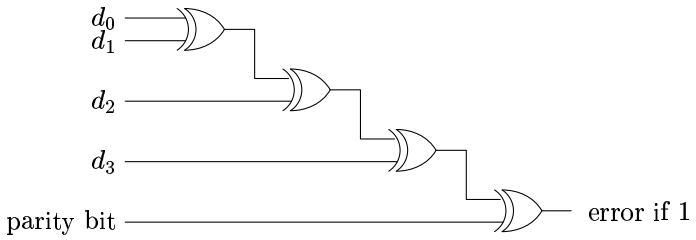

Figure 5.5. Logic diagram of a parity checker for 4-bit data (d_0, d_1, d_2, d_3).

Any computed parity bit that does not match the stored parity bit indicates that there was at least one bit error in the corresponding byte of data, or in the parity bit itself. An error signal, called *non-maskable interrupt*, is sent to the CPU to indicate that the memory data is not valid and to instruct the processor to immediately halt.

All operations related to the error-detection (encoding, decoding, comparison) are done by the memory control logic on the mother-board, in the chip set, or, for some systems, in the CPU. The memory itself only stores the parity bits, just as it stores the data bits. Therefore, parity checking does not slow down the operation of the memory. The parity bit generation and checking is done in parallel with the writing and reading of the memory using the logic which is much faster that the memory itself. Nothing in the system waits for a "no error" signal from the parity checker. The system only performs an action of interrupt when it finds an error.

Example 5.1. Suppose the data which is written in the memory is [0110110] and odd-parity code is used. Then the check bit 1 is stored along with the data to make the overall parity odd, i.e. the codeword [01101101]. Suppose that the codeword read out of the memory is [01111101]. The re-computed parity is 0. Because the re-computed parity disagree with the stored parity, we know that an error has occurred.

Parity can only detect single bit errors and an odd number of bit errors. If an even number of bits are affected, the computed parity matches the stored parity, and the erroneous data is accepted with no error notification, possibly causing later some mysterious problems. Studies have shown that approximately 98%

of all memory errors are single-bit errors. Thus, protecting a memory by a parity code is an inexpensive and efficient technique. For example, 1 GByte dynamic random access (DRAM) memory with a parity code has a failure rate of 0.7 failures per year. If the same memory uses a single-error correction double-error detection Hamming code, requiring 7 check bits in a 32-bit wide memory system, then the failure rate reduces to 0.03 failures per year. An error correcting memory is typically slower than a non-correcting one, due to the error correcting circuitry. Depending on the application, 0.7 failures per year may be viewed as an acceptable level of risk, or not.

A modification of the parity code is the *horizontal and vertical parity code*, which arranges the data in a 2-dimensional array and add one parity bit on each row and one parity bit on each column. Such a technique is useful for correcting single bit errors within a block of data words, however, it may fail correcting multiple errors.

4. Linear codes

Linear codes provide a general framework for generating many codes, including the Hamming code. The discussion of linear codes requires some knowledge of linear algebra, which we first briefly review.

4.1 Basic notions

Field Z_2. A *field* Z_2 is the set $\{0,1\}$ together with two operations, addition "$+$" and multiplication "$\cdot$", such that the following properties are satisfied for all $a,b,c \in Z_2$:

1 Z_2 is closed under '$\cdot$" and "$+$", meaning that $a \cdot b \in Z_2$ and $a+b \in Z_2$.

2 $a+(b+c) = (a+b)+c$.

3 $a+b = b+a$

4 There exists an element $\mathbf{0}$ in Z_2 such that $a+\mathbf{0} = a$.

5 For each $a \in Z_2$, there exists an element $-a \in Z_2$ such that $a+(-a) = \mathbf{0}$.

6 $a \cdot (b \cdot c) = (a \cdot b) \cdot c$.

7 $a \cdot b = b \cdot a$.

8 There exists an element $\mathbf{1}$ in Z_2 such that $a \cdot 1 = 1 \cdot a = a$.

9 For each $a \in Z_2$, such that $a \neq \mathbf{0}$, there exists an element $a^{-1} \in Z_2$ such that $a \cdot a^{-1} = \mathbf{1}$.

10 $a \cdot (b+c) = a \cdot b + a \cdot c$.

It is easy to see that above properties are satisfied if "+" is defined as addition modulo 2, "·" is defined as multiplication modulo 2, $\mathbf{0} = 0$ and $\mathbf{1} = 1$. Throughout the chapter, we assume this definition of Z_2.

Vector space V^n. Let Z_2^n denote the set of all n-tuples containing elements from Z_2. For example, for $n = 3$, $Z_2^3 = \{000, 001, 010, 011, 100, 110, 111\}$.

A *vector space* V^n over a field Z_2 is subset of Z_2^n, with two operations, addition "+" and multiplication "·", such that the following axioms are satisfied for all $x, y, z \in V^n$ and all $a, b, c \in Z_n$:

1 V^n is closed under "+", meaning that $x + y \in V^n$.

2 $x + y = y + x$.

3 $x + (y + z) = (x + y) + z$.

4 There exists an element $\mathbf{0}$ in V^n such that $x + \mathbf{0} = x$.

5 For each $x \in V^n$, there exists an element $-x \in V^n$ such that $x + (-x) = \mathbf{0}$.

6 $a \cdot x \in V^n$.

7 There exists an element $\mathbf{1} \in Z_2$ such that $1 \cdot x = x$.

8 $a \cdot (x + y) = (a \cdot x) + (a \cdot y)$.

9 $(a + b) \cdot x = a \cdot x + b \cdot x$.

10 $(a \cdot b) \cdot x = a \cdot (b \cdot x)$.

A *subspace* is a subset of a vector space that is itself a vector space.

A set of vectors $\{v_0, \ldots, v_{k-1}\}$ is said to *span* a vector space V^n if any $v \in V^n$ can be written as $v = a_0 v_0 + a_1 v_1 + \ldots + a_{k-1} v_{k-1}$, where $a_0, \ldots, a_{k-1} \in Z_2$.

A set of vectors $\{v_0, \ldots, v_{k-1}\}$ of V^n is said to be *linearly independent* if $a_0 v_0 + a_1 v_1 + \ldots + a_{k-1} v_{k-1} = 0$ implies that $a_0 = a_1 = \ldots = a_{k-1} = 0$.

A *basis* is a set of vectors in a vector space V^n that are linearly independent and span V^n.

The *dimension* of a vector space is defined to be the number of vectors in its basis.

4.2 Definition of linear code

A (n, k) linear code over the field Z_2 is a k-dimensional subspace of V_n. In other words, a linear code of length n is a subspace of V^n which is spanned by k linearly independent vectors. All codewords can be written as a linear combination of the k basis vectors $\{v_0, \ldots, v_{k-1}\}$ as follows:

$$c = d_0 v_0 + d_1 v_1 + \ldots + d_{k-1} v_{k-1}$$

Since a different codeword is obtained for each different combination of coefficients $d_0, d_1, \ldots, d_{k-1}$, we obtain an easy method of encoding if we define $d = (d_0, d_1, \ldots, d_{k-1})$ as the data to be encoded.

Example 5.2. As an example, let us construct a $(4,2)$ linear code.

The data we are encoding are 2-bit words $\{[00],[01],[10],[11]\}$. These words need to be encoded so that the resulting 4-bit codewords form a two-dimensional subspace of V^4. To do this, we have to select two linearly independent vectors as a basis of the two-dimensional subspace. One possibility is to choose the vectors $v_0 = [1000]$ and $v_1 = [0110]$. They are linearly independent since neither is a multiple of the other.

To find the codeword, c, corresponding to the data word $d = [d_0 d_1]$, we compute the linear combination of the two basis vectors $\{v_0, v_1\}$ as $c = d_0 v_0 + d_1 v_1$. Thus, the data word $d = [11]$ is encoded to

$$c = 1 \cdot [1000] + 1 \cdot [0110] = [1110]$$

Recall, the "+" is defined as an XOR, so $1 + 1 = 0$.

Similarly, $[00]$ is encoded to

$$c = 0 \cdot [1000] + 0 \cdot [0110] = [0000]$$

$[01]$ is encoded to

$$c = 0 \cdot [1000] + 1 \cdot [0110] = [0110]$$

and $[10]$ is encoded to

$$c = 1 \cdot [1000] + 0 \cdot [0110] = [1000].$$

4.3 Generator matrix

The computations we performed can be formalized by introducing the *generator matrix G*, whose rows are the basis vectors v_0 through v_{k-1}. For instance, the generator matrix for the Example 5.2 is

$$G = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{bmatrix}. \tag{5.1}$$

The codeword c is a product of the generator matrix G and the data word d:

$$c = dG$$

Note, that in the Example 5.2 the first two bits of each codeword are exactly the same as the data bits, i.e. the code we have constructed is a separable code. Separable linear codes are easy to decode by truncating the last $n-k$ bits.

In general, separability can be achieved by insuring that basis vectors form a generating matrix of the form $[I_k A]$, where I_k is an identity matrix of size $k \times k$.

Note also that the code distance of the code in the Example 5.2 is one. Therefore, such a code cannot detect errors. It is possible to predict code distance by examining the basic vectors. Consider the vector $v_1 = [1000]$ from the Example 5.2. Since it is a basic vector, it is a codeword. A code is a subspace V^n and thus is itself a vector space. A vector space is closed under addition, thus $c + 1000$ also belongs to the code, for any codeword c. The distance between c and $c + 1000$ is 1, since they differ only in the first bit position. Therefore, a code with a code distance δ should have basis vectors of weight greater than or equal to δ.

Example 5.3. Consider the $(6,3)$ linear code spanned by the basic vectors $[100011], [010110]$ and $[001101]$. The generator matrix for this code is

$$G = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix}. \tag{5.2}$$

For example, the data word $d = [011]$ is encoded to

$$c = 0 \cdot [100011] + 1 \cdot [010110] + 1 \cdot [001101] = [011011].$$

Recall, the "+" is modulo 2, so $1 + 1 = 0$. Similarly, we can encode other data words. The resulting code is presented in Table 5.1. Each row of the table shows a data word $d = [d_1 d_2 d_3]$ and the corresponding codeword $c = [c_1 c_2 c_3 c_4 c_5 c_6]$.

data			codeword					
d_1	d_2	d_3	c_1	c_2	c_3	c_4	c_5	c_6
0	0	0	0	0	0	0	0	0
0	0	1	0	0	1	1	0	1
0	1	0	0	1	0	1	1	0
0	1	1	0	1	1	0	1	1
1	0	0	1	0	0	0	1	1
1	0	1	1	0	1	1	1	0
1	1	0	1	1	0	1	0	1
1	1	1	1	1	1	0	0	0

Table 5.1. Defining table for a $(6,3)$ linear code.

4.4 Parity check matrix

To detect errors in a (n,k) linear code, we use an $(n-k)\times n$ matrix H, called the *parity check matrix* of the code. The parity check matrix represents the parity of the codewords. The matrix H has the property that, for any codeword c, $H\cdot c^T = 0$. By c^T we denote a transponse of the vector c. Recall, that the transpose of an $n\times k$ matrix A is the $k\times n$ matrix obtained by defining the ith column of A^T to be the ith row of A.

The parity check matrix is related to the generator matrix by the equation

$$HG^T = 0$$

This equation implies that, if data d is encoded to a codeword dG using the generator matrix G, then the product of the parity check matrix and the encoded message is zero. This is true because

$$H(dG)^T = H(G^T d^T) = (HG^T)d^T = 0$$

If a generator matrix is of the form $G = [I_k A]$, then

$$H = [A^T I_{n-k}]$$

is a parity check matrix. This can be proved as follows:

$$HG^T = A^T I_k + I_{n-k}A^T = A^T + A^T = 0.$$

Example 5.4. Let us construct the parity matrix H for the generator matrix G given by (5.2). G is of the form $[I_3 A]$ where

$$A = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}.$$

So, we have

$$H = [A^T I_3] = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}. \tag{5.3}$$

4.5 Syndrome

Encoded data can be checked for errors by multiplying it by the parity check matrix:

$$s = Hc^T \tag{5.4}$$

The resulting k-bit vector s is called *syndrome*. If the syndrome is zero, no error has occurred. If s matches one of the columns of H, then a single-bit error

has occurred. The bit position of the error corresponds to the position of the matching column in H. For example, if the syndrome coincides with the second column of H, the error is in the second bit of the codeword. If the syndrome is not zero and is not equal to any of the columns of H, then a multiple-bit error has occurred.

Example 5.5. As an example, consider the data $d = [110]$ encoded using the $(6,3)$ linear code from the Example 5.3 as $c = dG = [110101]$. Suppose that an error occurs in the second bit of c, transforming it to $[100101]$. By multiplying this word by the parity check matrix (5.3), we obtain the syndrome $s = [110]$. The syndrome matches the second column of the parity check matrix H, indicating that the error has occurred in the second bit.

4.6 Constructing linear codes

As we showed in Section 2.6, for a code to be able to correct ε errors, its code distance should be at least $2\varepsilon + 1$. It is possible to ensure a given code distance by carefully selecting the parity check matrix and then by using it to construct the corresponding generator matrix. It is proved that a code has distance of at least δ if and only if every subset of $\delta - 1$ columns of its parity check matrix H are linearly independent. So, to have a code distance two, we must ensure that every column of the parity check matrix is linearly independent. This is equivalent to the requirement of not having a zero column, since the zero vector can never be a member of a set of linearly independent vectors.

Example 5.6. In the parity check matrix (5.1) of the code which we have constructed in Example 5.2, the first column is zero:

$$H = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Therefore, columns of H are linearly dependent and the code distance is one.

Let us modify H to construct a new code with the code distance of at least two. Suppose to replace the zero column by the column containing 1 in all its entries:

$$H = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix} = [A^T I_2].$$

So, now A is

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}.$$

and therefor G can be constructed as

$$G = [I_2 A^T] = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}.$$

Using this generator matrix, the data words are encoded as dG resulting in a code shown in Table 5.2.

data		codeword			
d_1	d_2	c_1	c_2	c_3	c_4
0	0	0	0	0	0
0	1	0	1	1	0
1	0	1	0	1	1
1	1	1	1	0	1

Table 5.2. Defining table for a $(4,2)$ linear code.

The code distance of the resulting $(4,2)$ code is two. So, this code could be used to detect single-bit errors.

Example 5.7. Let us construct a code with a minimum code distance three, capable of correcting single-bit errors. We apply an approach similar to the one in Example 5.6. First, we create a parity check matrix in the form $[A^T I_{n-k}]$, such that every pair of its columns is linearly independent. This can be achieved by ensuring that each column is non-zero and no column is repeated twice.

For, example if our goal is to construct a $(3,1)$ code, then the following matrix

$$H = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}.$$

has all its columns non-zero and no column are repeats twice. The matrix A^T in this case is

$$A^T = \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

So, $A = [11]$ and therefore G is

$$G = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}.$$

The resulting (3,1) code consists of two codewords, 000 and 111.

4.7 Hamming codes

Hamming codes are a family of linear codes. They are named after Richard W. Hamming, who developed the first single-error correcting Hamming code and its extended version, single-error correcting double-error detecting Hamming code in the early 1950's. These codes remain important until today.

Consider the following parity check matrix, corresponding to a $(7,4)$ Hamming code:

$$H = \begin{bmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}. \tag{5.5}$$

H has $n = 7$ columns of length $n-k = 3$. Note, that $7 = 2^{7-4} - 1$, so the columns of H represent all possible non-zero vectors of length 3. In general, the parity matrix of a (n,k) Hamming code is constructed as follows. For a given $n-k$, construct a binary $n-k \times 2^{n-k} - 1$ matrix H such that each non-zero binary $n-k$-tuple occurs exactly once as a column of H. Any code with such a check matrix is called *binary Hamming code*. The code distance of any binary Hamming code is 3, so a Hamming code is a single-error correcting code.

If the columns of H are permuted, the resulting code remains a Hamming code, since the new check matrix is a set of all possible non-zero $n-k$-tuples. Different parity check matrices can be selected to suit different purposes. For example, by permuting the columns of the matrix (5.5), we can get the following matrix:

$$H = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}. \tag{5.6}$$

This matrix is a parity check matrix for a different $(7,4)$ Hamming code. Note, that its column i contains a binary representation of the integer $i \in \{1,2,\ldots,2^{n-k}-1\}$. A check matrix satisfying this property is called *lexicographic parity check matrix*. The code corresponding to the matrix (5.5) has a generator matrix in standard form $G = [I_3 A^T]$. The code corresponding to the matrix (5.6) does not have a generator matrix in standard form.

For a Hamming code with lexicographic parity check matrix, a simple procedure for syndrome decoding can be applied, similar to the one discussed earlier in Section 4.5. To check a codeword x for errors, we first calculate the syndrome $s = Hx^T$. If s is zero, then no error has occurred. If s is not zero, then it is a binary representation of some integer $i \in \{1,2,\ldots,2^{n-k}-1\}$. Then, x is decoded assuming that a single error has occurred in the ith bit of x.

Example 5.8. Let us construct a $(7,4)$ Hamming code corresponding to the parity check matrix (5.5). H in the form $[A^T I_3]$ where

$$A^T = \begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}.$$

The generator matrix is of the form $G = [I_4 A]$, or

$$G = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix}.$$

Suppose the data to be encoded is $d = [1110]$. We multiply d by G to get the codeword $c = [1110001]$. Suppose that an error occurs in the last bit of c, transforming it to $[1110000]$. Before decoding this word, we first check it for errors by multiplying it by the parity check matrix (5.5). The resulting syndrome $s = [001]$ matches the last column of H, indicating that the error has occurred in the last bit. So, we correct $[1110000]$ to $[1110001]$ and then decode it by taking the first four bits as data $d = [1110]$.

Example 5.9. The generator matrix corresponding to the lexicographic parity check matrix (5.6) is given by:

$$G = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

So, the data $d = [d_0 d_1 d_2 d_3]$ is encoded as $[p_3 p_2 d_2 p_1 d_1 d_0 d_3]$ where p_1, p_2, p_3 are parity check bits defined by $p_1 = d_0 + d_1 + d_2$, $p_2 = d_0 + d_2 + d_3$ and $p_3 = d_1 + d_2 + d_3$. The addition is modulo 2.

The *information rate* of a $(7,4)$ Hamming code is $k/n = 4/7$. In general the rate of a (n,k) Hamming code is $k/(2^{n-k} - 1)$.

Hamming codes are widely used for DRAM error-correction. Encoding is usually performed on complete words, rather than individual bytes. As in the parity code case, when a word is written into a memory, the check bits are computed by a check bits generator. For instance, for a $(7,4)$ Hamming code from Example 5.9, the check bits are computed as $p_1 = d_0 + d_1 + d_2$, $p_2 = d_0 + d_2 + d_3$ and $p_3 = d_1 + d_2 + d_3$, using a tree of XOR gates.

When the word is read back, check bits are recomputed and the syndrome is generated by taking an XOR of the read and recomputed check bits. If the syndrome is zero, no error has occurred. If the syndrome is non-zero, it is used to locate the faulty bit by comparing it to the columns of H. This can be implemented either in hardware or in software. If an (n,k) Hamming code with a lexicographic parity check matrix is used, then the error correction can be implemented using a decoder and XOR gates. If the syndrome $s = i$, $i \in \{1, 2, \ldots, 2^{n-k} - 1\}$, the ith bit of the word is faulty.

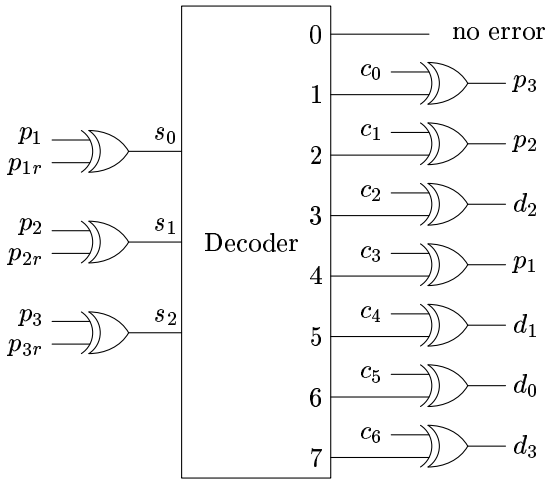

Figure 5.6. Error-correcting circuit for an $(7,4)$ Hamming code with a lexicographic parity check matrix.

An example of error correction for $(7,4)$ Hamming code from Example 5.8 is shown in Figure 5.6. The first level of XOR gates compares read check bits p_r with recomputed ones p. The result of this comparison is the syndrome $[s_0 s_1 s_2]$, which is fed into the decoder. For the syndrome $s = i$, $i \in \{0,1,\ldots,7\}$, the ith output of the decoder is high. The second level of XOR gates complements the ith bit of the word, thus correcting the error.

Often, the extended Hamming code rather than the regular Hamming code is used, which allows for not only the that single-bit error correction, but also double-bit errors detection. We describe this code in the next section.

4.8 Extended Hamming codes

The code distance of a Hamming code is three. If we add a parity check bit to every codeword of a Hamming code, then the code distance increases to four. The resulting code is called *extended Hamming code*. It can correct single-bit errors and detect double-bit errors.

The parity check matrix for an extended (n,k) Hamming code can be obtained by first adding a zero column in front of a lexicographic parity check matrix of an (n,k) Hamming code, and then by attaching a row consisting of all 1's as the $n-k+1$th row of the resulting matrix. For example, the matrix H for an extended $(1,1)$ Hamming code is given by

$$H = \left[\begin{array}{c|c} 0 & 1 \\ \hline 1 & 1 \end{array}\right].$$

The matrix H for an extended $(3,2)$ Hamming code is given by

$$H = \left[\begin{array}{c|ccc} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ \hline 1 & 1 & 1 & 1 \end{array}\right].$$

The matrix H for an extended $(7,4)$ Hamming code is given by

$$H = \left[\begin{array}{c|ccccccc} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ \hline 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{array}\right].$$

If $c = [c_1, c_2, \ldots, c_n]$ is a codeword from an (n,k) Hamming code, then $c' = [c_0, c_1, c_2, \ldots, c_n]$ is the corresponding extended codeword, where $c_0 = \sum_{i=1}^{n} c_i$ is the parity bit.

5. Cyclic codes

Cyclic codes are a special class of linear codes. Cyclic codes are used in applications where burst errors can occur, in which a group of adjacent bits is affected. Such errors are typical in digital communication as well as in storage devices, such as discs and tapes. A scratch on a compact disk is one example of a burst error. Two important classes of cyclic codes which we will consider are *cyclic redundancy check* (CRC) codes, used in modems and network protocols and *Reed-Solomon codes*, applied in satellite communication, wireless communication, compact disk players and DVDs.

5.1 Definition

A linear code is called cyclic if $[c_{n-1}c_0c_1c_2\ldots c_{n-2}]$ is a codeword whenever $[c_0c_1c_2\ldots c_{n-2}c_{n-1}]$ is also a codeword. So, any end-around shift of a codeword of a cyclic code produces another codeword.

When working with cyclic codes, it is convenient to think of words as polynomials rather than vectors. For a binary cyclic code, the coefficients of the polynomials are 0 or 1. For example, a data word $[d_0d_1d_2\ldots d_{k-1}d_k]$ is represented as a polynomial

$$d_0 \cdot x^0 + d_1 \cdot x^1 + d_2 \cdot x^2 + \ldots + d_{k-1} \cdot x^{k-1} + d_k \cdot x^k$$

where addition and multiplication is in the field Z_2, i.e modulo 2. The *degree* of a polynomial is equal to its highest exponent. For example, a word [1011] corresponds to the polynomial $1 \cdot x^0 + 0 \cdot x^1 + 1 \cdot x^2 + 1 \cdot x^3 = 1 + x^2 + x^3$ (least significant bit on the left). The degree of this polynomial is 3.

Before continuing with cyclic codes, we first review the basics of polynomial arithmetics, necessary for the understanding of encoding and decoding algorithms.

5.2 Polynomial manipulation

In this section we consider examples of polynomial multiplication and division. All the operations are carried out in the field Z_2, i.e. modulo 2.

Example 5.10. Compute $(1+x+x^3)\cdot(1+x^2)$.

$$(1+x+x^3)\cdot(1+x^2) = 1+x+x^3+x^2+x^3+x^5 = 1+x+x^2+x^5$$

Note, that $x^3+x^3=0$, since addition is modulo 2.

Example 5.11. Compute $(1+x^3+x^5+x^6)/(1+x+x^3)$.

$$\begin{array}{r|l}
x^3+x+1 & x^3+x^2+x+1 \\
\hline
 & x^6+x^5+x^3+1 \\
 & x^6+x^4+x^3 \\
\hline
 & x^5+x^4+1 \\
 & x^5+x^3+x^2 \\
\hline
 & x^4+x^3+x^2+1 \\
 & x^4+x^2+x \\
\hline
 & x^3+x+1 \\
 & x^3+x+1 \\
\hline
 & 0
\end{array}$$

So, $1+x+x^3$ divides $1+x^3+x^5+x^6$ without a reminder and the result is $1+x+x^2+x^3$.

For the decoding algorithm we also need to know how to perform arithmetic modulo $p(x)$, where $p(x)$ is a polynomial. To find $f(x)$ mod $p(x)$, we divide $f(x)$ by $p(x)$ and take the remainder.

Example 5.12. Compute $(1+x^2+x^5)$ mod $(1+x+x^3)$.

$$\begin{array}{r|l}
x^3+x+1 & x^3+x+1 \\
\hline
 & x^5+x^2+1 \\
 & x^5+x^3+x^2 \\
\hline
 & x^3+1 \\
 & x^3+x+1 \\
\hline
 & x
\end{array}$$

So, $(1+x^2+x^5)$ mod $(1+x+x^3)=x$.

5.3 Generator polynomial

To encode data in a cyclic code, the polynomial representing the data is multiplied by a polynomial known as *generator polynomial*. The generator

polynomial determines the properties of the resulting cyclic code. For example, suppose we encode the data [1011] using the generator polynomial $g(x) = 1+x+x^3$ (least significant bit on the left). The polynomial representing the data is $d(x) = 1+x^2+x^3$. By computing $g(x) \cdot d(x)$, we get $1+x+x^2+x^3+x^4+x^5+x^6$. So, the codeword corresponding to the data [1011] is [1111111].

The choice of the generator polynomials is guided by the property that $g(x)$ is the generator polynomial for a linear cyclic code of length n if and only if $g(x)$ divides $1+x^n$ without a reminder.

If n is the length of the codeword, then the length of the encoded data word is $k = n - deg(g(x))$ where $deg(g(x))$ denotes the degree of the generator polynomial $g(x)$. A cyclic code with a generator polynomial of degree $n-k$ is called (n,k) cyclic code. An (n,k) cyclic code can detect burst errors affecting $n-k$ bits or less.

Example 5.13. Find a generator polynomial for a code of length $n = 7$ for encoding data of length $k = 4$.

We are looking for a polynomial of degree $7-4 = 3$ which divides $1+x^7$ without a reminder. The polynomial $1+x^7$ can be factored as $1+x^7 = (1+x+x^3)(1+x^2+x^3)(1+x)$, so we can choose either $g(x) = 1+x+x^3$ or $g(x) = 1+x^2+x^3$. Table 5.3 shows the cyclic code generated by $g(x) = 1+x+x^3$. Since $deg(g(x)) = 3$, 3-bit burst errors can be detected by this code.

Let C be an (n,k) cyclic code generated by the generator polynomial $g(x)$. Codewords $x^i g(x)$ are basis for C, since every codeword

$$d(x)g(x) = d_0 g(x) + d_1 x g(x) + \ldots d_{k-1} x^{k-1} g(x)$$

is a linear combination of $x^i g(x)$. So, the following matrix G with rows $x^i g(x)$ is a generator matrix for C:

$$G = \begin{bmatrix} g(x) \\ xg(x) \\ \ldots \\ x^{k-2}g(x) \\ x^{k-1}g(x) \end{bmatrix} = \begin{bmatrix} g_0 & g_1 & \ldots & g_{n-k} & 0 & 0 & \ldots & 0 \\ 0 & g_0 & g_1 & \ldots & g_{n-k} & 0 & \ldots & 0 \\ & & & \ldots & & & & \\ 0 & 0 & 0 & g_0 & g_1 & \ldots & g_{n-k} & 0 \\ 0 & 0 & 0 & 0 & g_0 & g_1 & \ldots & g_{n-k} \end{bmatrix}.$$

Every row of G is a right cyclic shift of the first row. This generator matrix leads to a simple encoding algorithm using polynomial multiplication by $g(x)$.

Example 5.14. If C is a binary cyclic code with the generator polynomial $g(x) = 1+x+x^3$, then the generator matrix is given by:

$$G = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix}.$$

data				codeword						
d_0	d_1	d_2	d_3	c_0	c_1	c_2	c_3	c_4	c_5	c_6
0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1	1	0	1
0	0	1	0	0	0	1	1	0	1	0
0	0	1	1	0	0	1	0	1	1	1
0	1	0	0	0	1	1	0	1	0	0
0	1	0	1	0	1	1	1	0	0	1
0	1	1	0	0	1	0	1	1	1	0
0	1	1	1	0	1	0	0	0	1	1
1	0	0	0	1	1	0	1	0	0	0
1	0	0	1	1	1	0	0	1	0	1
1	0	1	0	1	1	1	0	0	1	0
1	0	1	1	1	1	1	1	1	1	1
1	1	0	0	1	0	1	1	1	0	0
1	1	0	1	1	0	1	0	0	0	1
1	1	1	0	1	0	0	0	1	1	0
1	1	1	1	1	0	0	1	0	1	1

Table 5.3. Defining table for (7,4) with the generator polynomial $g(x) = 1+x+x^3$.

5.4 Parity check polynomial

Given a cyclic code C with the generator polynomial $g(x)$, the polynomial $h(x)$ determined by

$$g(x)h(x) = 1+x^n$$

is the check polynomial of C. Since the codewords of a cyclic code are multiples of $g(x)$, for every codeword $c(x) \in C$, it holds that

$$c(x)h(x) = d(x)g(x)h(x) = d(x)(1+x^n) = 0 \bmod 1+x^n.$$

A *parity check matrix* H contains as its first row the coefficient of $h(x)$, starting from the most significant one. Every following row of H is a right cyclic shift of the first row.

$$H = \begin{bmatrix} h_k & h_{k-1} & \dots & h_0 & 0 & 0 & \dots & 0 \\ 0 & h_k & h_{k-1} & \dots & h_0 & 0 & \dots & 0 \\ & & & \dots & & & & \\ 0 & \dots & 0 & h_k & h_{k-1} & \dots & h_0 & 0 \\ 0 & \dots & 0 & 0 & h_k & h_{k-1} & \dots & h_0 \end{bmatrix}.$$

Example 5.15. Suppose C is a binary cyclic code with the generator polynomial $g(x) = 1 + x + x^3$. Let us compute its check polynomial.

There are three factors of $1 + x^7$: $1 + x, 1 + x + x^3$ and $1 + x^2 + x^3$. Thus, $h(x) = (1 + x^2 + x^3)(1 + x) = 1 + x + x^2 + x^4$. The corresponding parity check matrix is given by

$$H = \begin{bmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{bmatrix}. \tag{5.7}$$

5.5 Syndrome polynomial

Since cyclic codes are linear codes, we can use the parity check polynomial to detect errors which might have possibly occurred in a codeword $c(x)$ during data transmission or storage. We define

$$s(x) = h(x)c(x) \mod 1 + x^n$$

to be the *syndrome polynomial*. Since $g(x)h(x) = 1 + x^n$, syndrome can be computed by dividing the $c(x)$ by $g(x)$. If the reminder $s(x)$ is zero, then $c(x)$ is a codeword. Otherwise, there is an error in $c(x)$.

5.6 Implementation of polynomial division

The polynomial division can be implemented by linear feedback shift registers (LFSR). The logic diagram of an LFSR for the generator polynomial of degree $r = n - k$ is shown in Figure 5.7. It consists of a simple shift register and binary-weighted modulo 2 sums with feedback connections. Weights g_i, $i \in \{0, 1, \ldots, r-1\}$, are the coefficients of the generator polynomial $g(x) = g_0x^0 + g_2x^1 + \ldots g_nx^n$. Each g_i is either 0, meaning "no connection", or 1, meaning "connection". An exception is g_r which is always 1 and therefore is always connected.

If the input polynomial is $c(x)$, then the LFSR divides $c(x)$ by the generator polynomial $g(x)$, resulting in the quotient $d(x)$ and the reminder $s(x)$. The coefficients $[s_0s_1 \ldots s_r]$ of the syndrome polynomial are contained in the register after the division is completed. If syndrome is zero, then $c(x)$ is a codeword and $d(x)$ is valid data. If $[s_0s_1 \ldots s_r]$ matches one of the columns of parity check matrix H, then a single-bit error has occurred. The bit position of the error corresponds to the position of the matching column in H, so the error can be corrected. If the syndrome is not zero and is not equal to any of the columns of H, then a multiple-bit error has occurred which cannot be corrected.

Example 5.16. As an example, consider the logic circuit shown in Figure 5.8. It implements LFSR for the generator polynomial $g(x) = 1 + x + x^3$. Let s_i^+

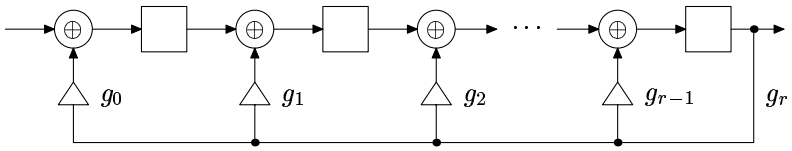

Figure 5.7. Logic diagram of a Linear Feedback Shift Register (LFSR).

denotes the next state value of the register cell s_i. Then, the state values of the LFSR in Figure 5.8 are given by

$$\begin{aligned} s_0^+ &= s_2 + c(x) \\ s_1^+ &= s_0 + s_2 \\ s_2^+ &= s_1 \end{aligned}$$

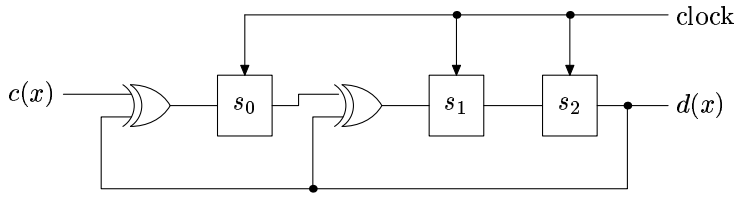

Figure 5.8. Implementation of the decoding circuit for a cyclic code with generator polynomial $g(x) = 1 + x + x^3$.

Suppose the word to be decoded is $[1010001]$, i.e. $1 + x^2 + x^6$. Table 5.4 shows the values of the register. This word is fed into the LFSR with the most significant bit first. The first bit of the quotient (the most significant one) appears at the output at the 4th clock cycle. In general, the first bit of the quotient comes out at the clock cycle $r+1$ for an LFSR of size $r = n-k$. After the division is completed at cycle 7 (cycle n for the general case), the state of the register is $[000]$, so $[1010001]$ is a codeword and the quotient $[1101]$ is valid data. We can verify the obtained result by dividing $1 + x^2 + x^6$ by $1 + x + x^3$. The quotient is $1 + x + x^3$, which is indeed $[1101]$.

Example 5.17. Next, suppose that a single-bit error has occurred in the 4th bit of codeword $[1010001]$, and a word $[1011001]$ is received instead. Table 5.5 illustrates the decoding process. As we can see, after the division is completed, the registers contain the reminder $[110]$, which matches the 4th column of the parity check matrix H in (5.7).

clock period	input $c(x)$	register state s_0	s_1	s_2	output $d(x)$
0		0	0	0	
1	1	1	0	0	0
2	0	0	1	0	0
3	0	0	0	1	0
4	0	1	1	0	1
5	1	1	1	1	0
6	0	1	0	1	1
7	1	0	0	0	1

Table 5.4. Register values of the circuit in Figure 5.8 for the input [1010001].

clock period	input $c(x)$	register state s_0	s_1	s_2	output $d(x)$
0		0	0	0	
1	1	1	0	0	0
2	0	0	1	0	0
3	0	0	0	1	0
4	1	0	1	0	1
5	1	1	0	1	0
6	0	1	0	0	1
7	1	1	1	0	0

Table 5.5. Register values of the circuit in Figure 5.8 for the input [1011001].

5.7 Separable cyclic codes

The cyclic codes that we have studied so far were not separable. It is possible to construct a separable cyclic code by applying the following technique.

First, we take the data $d = [d_0 d_1 \dots d_{k-1}]$ to be encoded and shift it right by $n-k$ positions:

$$[0, 0, \dots, 0, d_0, d_1, \dots, d_{k-1}]$$

Shifting the vector d right by $n-k$ positions corresponds to multiplying the data polynomial $d(x)$ by term x^{n-k}:

$$d(x)x^{n-k} = d_0 x^{n-k} + d_1 x^{n-k+1} + \dots + d_{k-1}x^{n-1}$$

Next, we employ the division algorithm to write

$$d(x)x^{n-k} = q(x)g(x) + r(x)$$

where $q(x)$ is a quotient and $r(x)$ is a reminder of division of $d(x)x^{n-k}$ by the generator polynomial $g(x)$. The reminder $r(x)$ has degree less than $n-k$, i.e. it is of type

$$[r_0, r_1, \ldots, r_{n-k-1}, 0, 0, \ldots, 0]$$

By moving $r(x)$ from the right hand side of the equation to the left hand side of the equation we get:

$$d(x)x^{n-k} + r(x) = q(x)g(x)$$

Recall, that "$-$" is equivalent to "$+$" in Z_2. Since the left hand side of this equation is equal to a multiple of $g(x)$, it is a codeword. This codeword has the form

$$[r_0, r_1, \ldots, r_{n-k-1}, d_0 d_1 \ldots d_k]$$

So, we have obtained a codeword in which the data is separated from the check bits.

Example 5.18. Let us demonstrate a systematic encoding for a (7,4) code with the generator polynomial $g(x) = 1 + x + x^3$. Let $d(x) = x + x^3$, i.e. $[0101]$.

First, we compute $x^{n-k}d(x) = x^3(x + x^3) = x^4 + x^6$. Next, we employ the division algorithm:

$$x^4 + x^6 = (1 + x^3)(1 + x + x^3) + (x + 1)$$

So, the resulting codeword is

$$c(x) = d(x)x^{n-k} + r(x) = 1 + x + x^4 + x^6$$

i.e. $[1100101]$. We can easily separate the data part of the codeword, it is contained in the last four bits.

Example 5.19. Suppose C is a binary separable cyclic code with the generator polynomial $g(x) = 1 + x + x^3$. Compute its generator and parity check matrices.

Consider the parity check matrix (5.7). To obtain a separable code, we need to permute its columns to bring it to the form $H = [A^T I_{n-k}]$. One of the solutions is:

$$H = \begin{bmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{bmatrix}.$$

The corresponding generator matrix $G = [I_k A]$ is:

$$G = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix}.$$

Since the encoding of a separable cyclic code involves division, it can be implemented using an LFSR identical to the one used for decoding. The multiplication by x^{n-k} is done by shifting the data right by $n-k$ positions. After the last bit of $d(x)$ has been fed in, the LFSR contains the reminder of division of input polynomial by $g(x)$. By subtracting this reminder from $x^{n-k}d(x)$ we obtain the encoded word.

5.8 CRC codes

Cyclic redundancy check (CRC) codes are separable cyclic codes with specific generator polynomials, chosen to provide high error detection capability for data transmission and storage. Common generator polynomials for CRC are:

CRC-16: $1+x^2+x^{15}+x^{16}$
CRC-CCITT: $1+x^5+x^{12}+x^{16}$
CRC-32:$1+x+x^2+x^4+x^7+x^8+x^{10}+x^{11}+x^{12}+x^{16}+x^{22}+x^{23}+x^{26}+x^{32}$

CRC-16 and CRC-CCITT are widely used in modems and network protocols in the USA and Europe, respectively, and give adequate protection for most applications. An attractive feature of CRC-16 and CRC-CCITT is the small number of non-zero terms in their polynomials (just four). This is an advantage because the LFSR required to implement encoding and decoding is simpler for generator polynomials with a smaller number of terms. Applications that need extra protection, such as Department of Defense applications, use CRC-32.

The encoding and decoding is done either in software, or in hardware, using the procedure from Section 5.7. To perform an encoding, the data polynomial is first shifted right by $deg(g(x))$ bit positions, and then divided by the generator polynomial. The coefficients of the remainder form the check bits of the CRC codeword. The number of check bits is equal to the degree of the generator polynomial. So, a CRC detects all burst errors of length less or equal to $deg(g(x))$. A CRC also detects many errors which are larger than $deg(g(x))$. For example, apart from detecting all burst errors of length 16 or less, CRC-16 and CRC-CCITT are also capable to detect 99.997% of burst errors of length 17 and 99.9985 burst errors of length 18.

5.9 Reed-Solomon codes

Reed-Solomon (RS) codes are a class of separable cyclic codes used to correct errors in a wide range of applications including storage devices (tapes, compact discs, DVDs, bar-codes), wireless communication (cellular telephones, microwave links), satellite communication, digital television, high-speed modems (ADSL, xDSL).

The encoding for Reed-Solomon codes is done similarly to the procedure described in Section 5.7. The codeword is computed by shifting the data right $n-k$ positions, dividing it by the generator polynomial and then adding the obtained reminder to the shifted data. A key difference is that groups of m bits rather than individual bits are used as symbols of the code. Usually $m = 8$, i.e. a byte. The theory behind is a field $Z_2^m$of degree m over $\{0,1\}$. The elements of such a field are m-tuples of 0 and 1, rather than just 0 and 1.

An encoder for an Reed-Solomon code takes k data symbols of s bits each and computes a codeword containing n symbols of m bits each. The maximum codeword length is related to m as $n = 2^m - 1$. A Reed-Solomon code can correct up to $\lfloor n-k \rfloor /2$ symbols that contain errors.

For example, a popular Reed-Solomon code is RS(255,223) where symbols are a byte (8-bit) long. Each codeword contains 255 bytes, of which 223 bytes are data and 32 bytes are check symbols. So, $n = 255, k = 223$ and therefore this code can correct up to 16 bytes containing errors. Note, that each of these 16 bytes can have multiple bit errors.

The decoding of Reed-Solomon codes is performed using an algorithm designed by Berlekamp. The popularity of Reed-Solomon codes is due to a large extent to the efficiency this algorithm. Berlekamp's algorithm was used by Voyager II for transmitting pictures of the outer space back to Earth. It is also a basis for decoding CDs in players. Many additional improvements were done over the years to make Reed-Solomon code practical. Compact discs, for example, use a modified version of RS code called *cross-interleaved* Reed-Solomon code.

6. Unordered codes

Unordered codes are designed to detect unidirectional errors. A *unidirectional error* is an error which changes either 0's of the word to 1, or 1's of the word to 0, but not both. An example of a unidirectional error is an error changing a word [1011000] to the word [0001000]. It is possible to apply a special design technique to ensure that most of the faults occurring in a logic circuit cause only unidirectional errors on the output. For example, consider the logic circuit shown in Figure 5.9. If a single stuck-at fault occurs at any of the lines in the circuit, it will cause a unidirectional error in the output word $[f_1 f_2 f_3]$.

The name of *unordered codes* originates from the following. We say that two binary n-tuples $x = (x_1, \ldots, x_n)$ and $y = (x_1, \ldots, x_n)$ are *ordered* if either $x_i \leq y_i$ for all $i \in \{1, 2, \ldots, n\}$, or $x_i \geq y_i$ for all i. For example if $x = [0101]$ and $y = [0000]$ then x and y are ordered, namely $x \geq y$. A unordered code is a code satisfying the property that any two of its codewords are unordered.

The ability of unordered codes to detect all unidirectional errors is directly related to the above property. A unidirectional error always changes a word

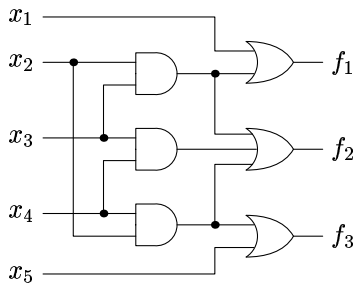

Figure 5.9. Logic diagram of a circuit in which any single stuck-at fault cause a unidirectional error on the output.

x to a word y which is either smaller or greater than x. A unidirectional error cannot change x to a word which is not ordered with x. Therefore, if any two of its codewords of a code are unordered, then a unidirectional error will never map a codeword to another codeword, and thus will be detected.

In this section we describe two unordered codes: m-of-n codes and Berger codes.

6.1 *M*-of-*n* codes

A m-of-n code consists of all n-bit words with exactly m 1's. Any k-bit unidirectional error forces the affected codeword to have either $m+k$ of $m-k$ 1's, and thus detected.

An easy way to construct an m-of-n code is to take the original k bits of data and append k bits so that the resulting $2k$-bit code word has exactly k 1's. For example, the 3-of-6 code is shown in Table 5.6. All codewords have exactly three 1's.

An obvious disadvantage of an $2k$-of-k is its low information rate of 1/2. An advantage of this code is its separability, which simplifies the encoding and decoding procedures. A more efficient m-of-n code, with higher information rate can be constructed, but then the separable nature of the code is usually lost. Non-separability makes the encoding, decoding and error detection procedures more difficult.

6.2 Berger codes

Check bits in a Berger code represent the number of 1's in the data word. A Berger code of length n has k data bits and m check bits, where $m = \lceil \log_2(k+1) \rceil$ and $n = k+m$. A codeword is created by complementing the m-bit binary representation of the number of 1's in the encoded word. An example of Berger code for 4-bit data is shown in Table 5.7.

data			codeword					
d_0	d_1	d_2	c_0	c_1	c_2	c_3	c_4	c_5
0	0	0	0	0	0	1	1	1
0	0	1	0	0	1	1	1	0
0	1	0	0	1	0	1	0	1
0	1	1	0	1	1	1	0	0
1	0	0	1	0	0	0	1	1
1	0	1	1	0	1	0	1	0
1	1	0	1	1	0	0	0	1
1	1	1	1	1	1	0	0	0

Table 5.6. Defining table for 3-of-6 code.

data				codeword						
d_0	d_1	d_2	d_3	c_0	c_1	c_2	c_3	c_4	c_5	c_6
0	0	0	0	0	0	0	0	1	1	1
0	0	0	1	0	0	0	1	1	1	0
0	0	1	0	0	0	1	0	1	1	0
0	0	1	1	0	0	1	1	1	0	1
0	1	0	0	0	1	0	0	1	1	0
0	1	0	1	0	1	0	1	1	0	1
0	1	1	0	0	1	1	0	1	0	1
0	1	1	1	0	1	1	1	1	0	0
1	0	0	0	1	0	0	0	1	1	0
1	0	0	1	1	0	0	1	1	0	1
1	0	1	0	1	0	1	0	1	0	1
1	0	1	1	1	0	1	1	1	0	0
1	1	0	0	1	1	0	0	1	0	1
1	1	0	1	1	1	0	1	1	0	0
1	1	1	0	1	1	1	0	1	0	0
1	1	1	1	1	1	1	1	0	1	1

Table 5.7. Defining table for Berger code for 4-bit data.

The primary advantages of a Berger code are that it is a separable code and that it detects all unidirectional multiple errors. It is shown that the Berger code is the most compact code for this purpose. The information rate of a Berger code for m-bit data is $k/(\lceil k+\lceil \log_2(k+1)\rceil\rceil)$. Table 5.8 shows how the information

rate grows as the size of the encoded data increases. For data of small size, the redundancy of a Berger code is high. However, as k increases, the number of check bits drops substantially. The Berger codes with $k = 2^m - 1$ are called *maximal length* Berger codes.

number of data bits	number or check bits	information rate
4	3	0.57
8	4	0.67
16	5	0.76
32	6	0.84
64	7	0.90
128	8	0.94

Table 5.8. Information rate of different Berger codes.

7. Arithmetic codes

Arithmetic codes are usually used for detecting errors in arithmetic operations, such as addition or multiplication. The data representing the operands, say b and c, is encoded before the operation is performed. The operation is carried out on the resulting codewords $A(b)$ and $A(c)$. After the operation, the codeword $A(b) * A(c)$ representing the result of the operation "$*$" is decoded and checked for errors.

An arithmetic code relies on the property of *invariance* with respect to the operation "$*$":

$$A(b * c) = A(b) * A(c)$$

Invariance guarantees that the operation "$*$" on codewords $A(b)$ and $A(c)$ gives us the same result as $A(b * c)$. So, if no error has occurred, decoding $A(b * c)$ gives us $b * c$, the result of the operation "$*$" on b and c.

Two common types of arithmetic codes are *AN*-codes and residue codes.

7.1 AN-codes

AN-code is the simplest representative of arithmetic codes. The codewords are obtained by multiplying data words N by some constant A. For example, if the data is of length two, namely [00], [01], [10], [11], then the $3N$-code is [0000], [0011], [0110], [1001]. Each codeword is computed by multiplying a data word by 3. And vice versa, to decode a codeword, we divide it by 3. If there is no reminder, no error has occurred. *AN*-codes are non-separable codes.

The *AN*-code is invariant with respect to addition and subtraction, but not to multiplication and division. For example, clearly $3(a \cdot b) \neq 3a \cdot 3b$ for all non-zero a, b.

The constant A determines the information rate of the code and its error detection capability. For binary codes, A should not be a power of two. This is because single-bit errors cause multiplication or division of the original codeword by 2^r, where r is the position of the affected bit. Therefore, the resulting word is a codeword and the error cannot be detected.

7.2 Residue codes

Residue codes are separable arithmetic codes which are created by computing a residue for data and appending it to the data. The residue is generated by dividing a data by an integer, called *modulus*. Decoding is done by simply removing the residue.

Residue codes are invariant with respect to addition, since

$$(b+c) \bmod m = b \bmod m + c \bmod m$$

where b and c are data words and m is modulus. This allows us to handle residues separately from data during addition process. The value of the modulus determines the information rate and the error detection capability of the code.

A variation of residue codes are *inverse residue code*, where an inverse of the residue, rather than the residue itself, is appended to the data. These codes are shown to have better fault detecting capabilities for common-mode faults.

8. Problems

5.1. Give an example of a binary code of length 4 and of size 6. How many words are contained in the codespace of your code?

5.2. Why is the separability of a code considered to be a desirable feature?

5.3. Define the information rate. How is the information rate related to redundancy?

5.4. What is the main difference in the objectives of encoding for the coding theory and the cryptography?

5.5. What is the maximum Hamming distance between two words in the codespace $\{0,1\}^4$?

5.6. Consider the code $C = \{01100101110, 10010110111, 01010011001\}$.

(a) What is the code distance of C?

(b) How many errors can be detected/corrected by code C?

5.7. How would you generalize the notions of Hamming distance and code distance to ternary codes using $\{0,1,2\}$ as valid symbols? Find a generalization which preserves the following two properties: To be able to correct ε-digit errors, a ternary code should have the code distance of at least $2\varepsilon+1$. To be able to detect ε-digit errors, the ternary code distance should be at least $\varepsilon+1$.

5.8. Prove that, for any $n>1$, a parity code of length n has code distance two.

5.9. (a) Construct an even parity code C for 3-bit data.

(b) Suppose the word (1101) is received. Assuming single bit error, what are the codewords that have possibly been transmitted?

5.10. Draw a gate-level logic circuit of an odd parity generation circuit for 5-bit data. Limit yourself to use of two-input gates only.

5.11. How would you generalize the notion of parity for ternary codes? Give an example of a ternary parity code for 3-digit data, satisfying your definition.

5.12. Construct the parity check matrix H and the generator matrix G for a linear code for 4-bit data which can:

(a) detect 1 error

(b) correct 1 error

(c) correct 1 error and detect one additional error.

5.13. Construct the parity check matrix H and the generator matrix G for a linear code for 5-bit data which can:

(a) detect 1 error

(b) correct 1 error

(c) correct 1 error and detect one additional error.

5.14. (a) Construct the parity check matrix H and the generator matrix G of a Hamming code for 11-bit data.

(a) Find whether you can construct a Hamming code for data of lengths $1,2$ and 3. Construct the parity check matrix H and the generator matrix G for whose lengths for which it is possible.

5.15. The parity code is a linear code, too. Construct the parity check matrix H and the generator matrix G for a parity code for 4-bit data.

5.16. Find the generator matrix for the (7,4) cyclic code C with the generator polynomial $1+x^2+x^3$. Prove that C is a Hamming code.

5.17. Find the generator matrix for the (15,11) cyclic code C with the generator polynomial $1+x+x^4$. Prove that C is a Hamming code.

5.18. Compute the check polynomial for the (7,4) cyclic code with the generator polynomial $g(x) = 1+x^2+x^3$.

5.19. Let C be and (n,k) cyclic code. Prove that the only burst errors of length $n-k+1$ that are codewords (and therefore not detectable errors) are shifts of scalar multiples of the generator polynomial.

5.20. Suppose you use a cyclic code generated by the polynomial $g(x) = 1+x+x^3$. You have received a word $c(x) = 1+x+x^4+x^5$. Check whether an error has occur during transmission.

5.21. Develop an LFSR for decoding of 4-bit data using the generator polynomial $g(x) = 1+x^4$. Show the state table for the word $c(x) = x^7+x^6+x^5+x^3+1$ (as the one in Table 5.4). Is $c(x)$ a valid codeword?

5.22. Construct a separable cyclic code for 4-bit data generated by the polynomial $g(x) = 1+x+x^4$. What code distance has the resulting code?

5.23. (a) Draw an LFSR decoding circuit for CRC codes with the following generator polynomials:

$$CRC-16 : 1+x^2+x^{15}+x^{16}$$

$$CRC-CCITT : 1+x^5+x^{12}+x^{16}$$

You may use "..." between the registers 2 and 15 in the 1st polynomial and 5 and 12 in the second, to make the picture shorter.

(b) Use the first generator polynomial for encoding the data $1+x^3+x^4$.

(c) Suppose that the error $1+x+x^2$ is added to the codeword you obtained in the previous task. Check whether this error will be detected or not.

5.24. Construct a Berger code for 3-bit data. What code distance has the resulting code?

5.25. Suppose we know that the original 4-bit data words will never include the word 0000. Can we reduce the number of check bits required for a Berger code and still cover all unidirectional errors?

5.26. Suppose we encoded 8-bit data using a Berger code.

(a) How many check bits are required?

(b) Take one codeword c and list all possible unidirectional errors which can affect c.

5.27. (a) Construct a 3*N* arithmetic code for 3-bit data.

(b) Give an example of a fault which is detected by such a code and an example a of fault which is not detected by such a code.

5.28. Consider the following code *C*:

0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	1	0
0	0	1	1	1	1
0	1	0	1	0	0
0	1	1	0	0	1
0	1	1	1	1	0
1	0	0	0	1	1

(a) What kind of code is this?

(b) Is it a separable code?

(c) What is the code distance of *C*?

(d) What kind of faults can it detect/correct?

(e) How are encoding and decoding done for this code? (describe in words)

(f) How error is detection done for this code? (describe in words)

5.29. Consider the following code *C*:

0	0	0	0	0	0
0	0	0	0	1	1
0	0	0	1	1	0
0	0	1	0	0	1
0	0	1	1	0	0
0	0	1	1	1	1
0	1	0	0	1	0
0	1	0	1	0	1
0	1	1	0	0	0
0	1	1	0	1	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	0	1	0	0
1	0	0	1	1	1
1	0	1	0	1	0
1	0	1	1	0	1

(a) What kind of code is this?

(b) Is it a separable code?

(c) What is the code distance of C?

(d) What kind of faults can it detect/correct?

(e) How are encoding and decoding done for this code? (describe in words)

(f) How is error detection done for this code? (describe in words)

5.30. Consider the following code:

0	0	0	1	1	1
0	0	1	1	1	0
0	1	0	1	0	1
0	1	1	1	0	0
1	0	0	0	1	1
1	0	1	0	1	0
1	1	0	0	0	1
1	1	1	0	0	0

(a) What kind of code is this?

(b) Is it a separable code?

(c) What is the code distance of C?

(d) What kind of faults can it detect/correct?

(e) Design a circuit (gate-level) for encoding of 3-bit data in this code. Your circuit should have 3 inputs for data bits and 6 outputs for codeword bits.

(f) How would you suggest to do error detection for this code? (describe in words).

<

5.31. Develop a scheme for active hardware redundancy (either standby sparing or pair-and-a-spare) employing error detection code of your choice for 1-bit error detection.

Chapter 6

TIME REDUNDANCY

1. Introduction

Space redundancy techniques discussed so far impact physical entities like cost, weight, size, power consumption, etc. In some applications extra time is of less importance than extra hardware.

Time redundancy is achieved by repeating the computation or data transmission and comparing the result to a stored copy of the previous result. If the repetition is done twice, and if the fault which has occurred is transient, then the stored copy will differ from the re-computed result, so the fault will be detected. If the repetition is done three or more times, a fault can be corrected. In this section, we show that time redundancy techniques can also be used for detecting permanent faults.

Apart from detection and correction of faults, time redundancy is useful for distinguishing between transient and permanent faults. If the fault disappears after the re-computation, it is assumed to be transient. In this case the hardware module is still usable and it would be a waste of resources to switch it off the operation.

2. Alternating logic

The alternating logic time redundancy scheme was developed by Reynolds and Metze in 1978. It has been applied to permanent fault detection in digital data transmission and in digital circuits.

Suppose the data is transmitted over a parallel bus as shown in Figure 6.1. At time t_0, the original data is transmitted. Then, the data is complemented and re-transmitted at time $t_0 + \Delta$. The two results are compared to check whether they are complements of each other. Any disagreement indicates a fault. Such a scheme is capable of detecting permanent stuck-at faults at the bus lines.

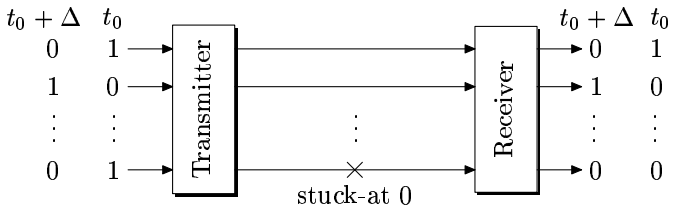

Figure 6.1. Alternating logic time redundancy scheme.

Alternating logic concept can be used for detecting fault in logic circuits which implement self-dual functions. A *dual* of a function $f(x_1, x_2, \ldots, x_n)$ is defined as

$$f_d(x_1, x_2, \ldots, x_n) = f'(x_1', x_2', \ldots, x_n'),$$

where $'$ denotes the complement. For example, a 2-variable AND $f(x_1, x_2) = x_1 \cdot x_2$ is dual of a 2-variable OR $(x_1, x_2) = x_1 + x_2$, and vice versa. A function in said to be *self-dual* if it is equal to its dual $f = f_d$. So, the value of a self-dual function f for the input assignment $x_1, x_2, \ldots, x_n$ equals to the value of the complement of f for the input assignment $x_1', x_2', \ldots, x_n'$. Examples of self-dual functions are sum and carry-out output functions of a full-adder (Figure 6.2). The sum $s(a, b, c_{in}) = a \oplus b \oplus c_{in}$, where "$\oplus$" is an XOR. The carry-out

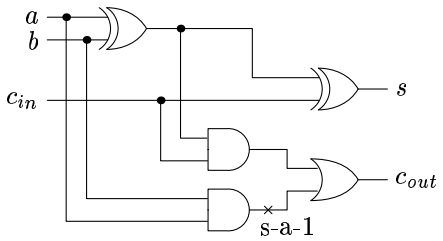

Figure 6.2. Logic diagram of a full-adder.

$c_{out}(a, b, c_{in}) = ab + (a \oplus b)c_{in}$. Table 6.1 shows the defining table for s and c_{out}. It is easy to see that the property $f(x_1, x_2, \ldots, x_n) = f'(x_1', x_2', \ldots, x_n')$ holds for both functions.

For a circuit implementing a self-dual function, the application of an input assignment $(x_1, x_2, \ldots, x_n)$ followed by the input assignment $(x_1', x_2', \ldots, x_n')$ should produce output values which are complements of each other, unless the circuit has a fault. So, a fault can be detected by finding an input assignment for which $f(x_1, x_2, \ldots, x_n) = f(x_1', x_2', \ldots, x_n')$. For example, a stuck-at-1 fault marked in Figure 6.2 can be detected by applying the input assignment $(a, b, c_{in}) = (100)$, followed by the complemented assignment (011). In a fault-free full-adder

a	b	c_{in}	s	c_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table 6.1. Defining table for a full-adder.

$s(100) = 1$, $c_{out}(100) = 0$ and $s(011) = 0$, $c_{out}(011) = 1$. However, in presence of the marked fault $s(100) = 1$, $c_{out}(100) = 1$ and $s(011) = 0$, $c_{out}(011) = 1$. Since $c_{out}(100) = c_{out}(011)$, the fault is detected.

If the function $f(x_1, x_2, \ldots, x_n)$ realized by the circuit is not self-dual, then it can be transformed to a self-dual function of $n+1$-variables, defined by

$$f_{sd} = x_{n+1} f + x'_{n+1} f_d$$

The new variable x_{n+1} is a control variable determining whether the value of f or f_d appears on the output. Clearly, such a function f_{sd} produces complemented values for complemented inputs. A drawback of this technique is that the circuit implementing f_{sd} can be twice as large as the circuit implementing f.

3. Recomputing with shifted operands

Recomputing with shifted operands (RESO) time redundancy technique was developed by Patel and Fung in 1982 for on-line fault detection in arithmetic logic units (ALUs) with bit-sliced organization.

At time t_0, the bit slice i of a circuit performs a computation. Then, the data is shifted left and the computation is repeated at time $t_0 + \delta$. The shift operand can be either arithmetic or logical shift. After the computation, the result is shifted right. The two results are compared. If there is no error, they are the same. Otherwise, they disagree in either the ith, or $(i-1)$th, or both bits.

The fault detection capability of RESO depends on the amount of shift. For example, for a bit-sliced ripple-carry adder, a 2-bit arithmetic shift is required to guarantee the fault detection. A fault in the ith bit of a slice can have one of the three effects:

1 The sum bit is erroneous. Then, the incorrect result differs form the correct one by either -2^i (if the sum is 0), or by $+2^i$ (if the sum is 1).

2 The carry bit is erroneous. Then, the incorrect result differs from the correct one by either -2^{i+1} (if the carry is 0), or by $+2^{i+1}$ (if the carry is 1).

3 Both, sum and carry bits, are erroneous. Then, we have four possibilities

- sum is 0, carry is 0: $-3\cdot 2^i$;
- sum is 0, carry is 1: $+2^i$;
- sum is 1, carry is 0: -2^i;
- sum is 1, carry is 1: $+3\cdot 2^i$;

Summarizing, if the operands are not shifted, then the erroneous result differs from the correct one by one of the following values: $\{0, 2^i, 2^{i+1}, 3\cdot 2^{i+1}\}$.

A similar analysis can be done to show that if the operands are shifted left by two bits, then the erroneous result differs from the correct one by one of the following values: $\{0, 2^{i-1}, 2^{i-2}, 3\cdot 2^{i-2}\}$. So, results of non-shifted and shifted computations cannot agree unless they are both correct.

A primary problem with RESO technique is the additional hardware required to store the shifted bits.

4. Recomputing with swapped operands

Recomputing with swapped operands (RESWO) is another time redundancy technique, introduced by Johnson in 1988. In RESWO, both operands are splitted into two halves. During the first computation, the operands are manipulated as usual. The second computation is performed with the lower and the upper halves of operands swapped.

RESWO technique can detect faults in any single bit slice. For example, consider a bit-sliced ripple-carry adder with n-bit operands. Suppose the lower half of an operand contains the bits from 0 to $r = n/2$ and the upper half contains the bits from $r+1$ to $n-1$. During the first computation, if the sum or carry bits from slice i are faulty, then the resulting sum differs from the correct one by 2^i and 2^{i+1}, respectively. If both, the sum and the carry are faulty, then the result differs from the correct one by $2^i 2^{i+1}$. So, before the operands' halves are swapped, a faulty bit slice i would cause the result to disagree from the correct result by one of the values $\{0, 2^i, 2^{i+1}, 2^i 2^{i+1}\}$. If $i \leq r$, the result of the re-computation with the lower and the upper halves of operands swapped differs from the correct result by one of the values $\{0, 2^{i-r}, 2^{i-r-1}, 2^{i-r}2^{i-r-1}\}$. This implies that the results of non-swapped and swapped computations cannot agree unless they are both correct.

5. Recomputing with duplication with comparison

Recomputing using duplication with comparison (REDWC) technique combines hardware redundancy with time redundancy. An n-bit operation is performed by using two $n/2$-bit devices twice. The operands are split into two

halves. First, the operation is carried out on the lower halves and their duplicates and the results are compared. This is then repeated for the upper halves of the operands.

As an example, consider how REDWC is performed on an n-bit full adder. First, lower and upper parts of the adder are used to compute the sum of the lower parts of the operands. A multiplexer is used to handle the carries at the boundaries of the adder. The results are compared and one of them is stored to represent the lower half of the final sum. The second computation is carried out on the upper parts of the operands. Selection of the appropriate half of the operands is performed using multiplexers.

REDWC technique allows to detect all single faults in one half of the adder, as long as both halves do not to become faulty in a similar manner or at the same time.

6. Problems

6.1. Give three examples of applications where time is less important than hardware.

6.2. Two independent methods for fault detection on busses are:

- the use of a parity bit,
- the use of alternating logic.

Neither of these methods has the capability of correcting an error. However, together these two methods can be used to correct any single permanent fault (stuck-at type). Explain how. Use an example to illustrate your algorithm.

6.3. Write a truth table for a 2-bit full adder and check whether the sum s and the carry out c_{out} are self-dual functions.

Chapter 7

SOFTWARE REDUNDANCY

Programs are really not much more than the programmer's best guess about what a system should do.

—Russel Abbot

1. Introduction

In this chapter, we discuss techniques for software fault-tolerance. In general, fault-tolerance in the software domain is not as well understood and mature as fault-tolerance in the hardware domain. Controversial opinions exist on whether reliability can be used to evaluate software. Software does not degrade with time. Its failures are mostly due to the activation of specification or design faults by the input sequences. So, if a fault exists in software, it will manifest itself first time when the relevant conditions occur. This makes the reliability of a software module dependent on the environment that generates the input to the module over time. Different environments might result in different reliability values. The Ariane 5 rocket accident is an example of how a piece of software, safe for the Ariane 4 operating environment, can cause a disaster in a new environment. As we described in Section 3.2, the Ariane 5 rocket exploded 37 seconds after its lift-off, due to complete loss of guidance and attitude information. The loss of information was caused by a fault in the software of the inertial reference system, resulted from violating the maximum floating point number assumption.

Many current techniques for software fault tolerance attempt to leverage the experience of hardware redundancy schemes. For example, software N-version programming closely resembles hardware N-modular redundancy. Recovery blocks use the concept of retrying the same operation in expectation that the problem is resolved after the second try. However, traditional hardware fault tolerance techniques were developed to fight permanent components faults pri-

marily, and transient faults caused by environmental factors secondarily. They do not offer sufficient protection against design and specification faults, which are dominant in software. By simply triplicating a software module and voting on its outputs we cannot tolerate a fault in the module, because all copies have identical faults. Design diversity technique, described in Section 3.3, has to be applied. It requires creation of diverse and equivalent specifications so that programmers can design software which do not share common faults. This is widely accepted to be a difficult task.

A software system usually has a very large number of states. For example, a collision avoidance system required on most commercial aircraft in the U.S., has 1040 states. Large number of states would not be a problem if the states exhibited adequate regularity to allow grouping them into equivalence classes. Unfortunately, software does not exhibit the regularity commonly found in digital hardware. The large number of states implies that only a very small part of the software system can be verified for correctness. Traditional testing and debugging methods are not feasible for large systems. The recent focus on using formal methods to describe the required characteristics of the software behavior promises higher coverage, however, due to their extremely large computational complexity formal methods are only applicable in specific applications. Due to incomplete verification, some design faults are not diagnosed and are not removed from the software.

Software fault-tolerance techniques can be divided into two groups: single-version and multi-version. Single version techniques aim to improve fault-tolerant capabilities of a single software module by adding fault detection, containment and recovery mechanisms to its design. Multi-version techniques employ redundant software modules, developed following design diversity rules. As in the hardware case, a number of possibilities has to be examined to determine at which level the redundancy needs to be provided and which modules are to be made redundant. The redundancy can be applied to a procedure, or to a process, or to the whole software system. Usually, the components which have high probability of faults are chosen to be made redundant. As in the hardware case, the increase in complexity caused by the redundancy can be quite severe and may diminish the dependability improvement, unless redundant resources are allocated in a proper way.

2. Single-version techniques

Single version techniques add to a single software module a number of functional capabilities that are unnecessary in a fault-free environment. The software structure and its actions are modified to be able to detect a fault, isolate it and prevent the propagation of its effect throughout the system. In this section, we consider how fault detection, fault containment and fault recovery are achieved in the software domain.

2.1 Fault detection techniques

As in the hardware case, the goal of fault detection in software is to determine that a fault has occurred within a system. Single-version fault tolerance techniques usually use various types of *acceptance tests* to detect faults. The result of a program is subjected to a test. If the result passes the test, the program continues its execution. A failed test indicates a fault. A test is most effective if it can be calculated in a simple way and if it is based on criteria that can be derived independently of the program application. The existing techniques include timing checks, coding checks, reversal checks, reasonableness checks and structural checks.

Timing checks are applicable to systems whose specification include timing constrains. Based on these constrains, checks can be developed to indicate a deviation from the required behavior. A *watchdog timer* is an example of a timing check. Watchdog timers are used to monitor the performance of a system and detect lost or locked out modules.

Coding checks are applicable to systems whose data can be encoded using information redundancy techniques. Cyclic redundancy checks can be used in cases when the information is merely transported from one module to another without changing it content. Arithmetic codes can be used to detect errors in arithmetic operations.

In some systems, it is possible to reverse the output values and to compute the corresponding input values. For such system, *reversal checks* can be applied. A reversal check compares the actual inputs of the system with the computed ones. A disagreement indicates a fault.

Reasonableness checks use semantic properties of data to detect fault. For example, a range of data can be examined for overflow or underflow to indicate a deviation from system's requirements.

Structural checks are based on known properties of data structures. For example, a number of elements in a list can be counted, or links and pointers can be verified. Structural checks can be made more efficient by adding redundant data to a data structure, e.g. attaching counts on the number of items in a list, or adding extra pointers.

2.2 Fault containment techniques

Fault containment in software can be achieved by modifying the structure of the system and by putting a set of restrictions defining which actions are permissible within the system. In this section, we describe four techniques for fault containment: modularization, partitioning, system closure and atomic actions.

It is common to decompose a software system into *modules* with few or no common dependencies between them. Modularization attempts to prevent

the propagation of faults by limiting the amount of communication between modules to carefully monitored messages and by eliminating shared resources. Before performing modularization, visibility and connectivity parameters are examined to determine which module possesses highest potential to cause system failure. The *visibility* of a module is characterized by the set of modules that may be invoked directly or indirectly by the module. The *connectivity* of a module is described by the set of modules that may be invoked directly or used by the module.

The isolation between functionally independent modules can be done by *partitioning* the modular hierarchy of a software architecture in horizontal or vertical dimensions. Horizontal partitioning separates the major software functions into independent branches. The execution of the functions and the communication between them is done using control modules. Vertical partitioning distributes the control and processing function in a top-down hierarchy. High-level modules normally focus on control functions, while low-level modules perform processing.

Another technique used for fault containment in software is *system closure*. This technique is based on the principle that no action is permissible unless explicitly authorized. In an environment with many restrictions and strict control (e.g. in prison) all the interactions between the elements of the system are visible. Therefore, it is easier to locate and remove any fault.

An alternative technique for fault containment uses *atomic actions* to define interactions between system components. An atomic action among a group of components is an activity in which the components interact exclusively with each other. There is no interaction with the rest of the system for the duration of the activity. Within an atomic action, the participating components neither import, nor export any type of information from non-participating components of the system. There are two possible outcomes of an atomic action: either it terminates normally, or it is aborted upon a fault detection. If an atomic action terminates normally, its results are correct. If a fault is detected, then this fault affects only the participating components. Thus, the fault containment area is defined and fault recovery is limited to the atomic action components.

2.3 Fault recovery techniques

Once a fault is detected and contained, a system attempts to recover from the faulty state and regain operational status. If fault detection and containment mechanisms are implemented properly, the effects of the faults are contained within a particular set of modules at the moment of fault detection. The knowledge of fault containment region is essential for the design of effective fault recovery mechanism.

2.3.1 Exception handling

In many software systems, the request for initiation of fault recovery is issued by *exception handling*. Exception handling is the interruption of the normal operation to handle abnormal responses. Possible events triggering the exceptions in a software module can be classified into three groups:

1 *Interface exceptions* are signaled by a module when it detects an invalid service request. This type of exception is supposed to be handled by the module that requested the service.

2 *Local exceptions* are signaled by a module when its fault detection mechanism detects a fault within its internal operations. This type of exception is supposed to be handled by the faulty module.

3 *Failure exceptions* are signaled by a module when it has detected that its fault recovery mechanism is unable to recover successfully. This type of exception is supposed to be handled by the system.

2.3.2 Checkpoint and restart

A popular recovery mechanism for single-version software fault tolerance is *checkpoint and restart*, also referred to as *backward error recovery*. As mentioned previously, most of the software faults are design faults, activated by some unexpected input sequence. These type of faults resemble hardware intermittent faults: they appear for a short period of time, then disappear, and then may appear again. As in the hardware case, simply restarting the module is usually enough to successfully complete its execution.

The general scheme of checkpoint and restart recovery mechanism is shown in Figure 7.1. The module executing a program operates in combination with an acceptance test block AT which checks the correctness of the result. If a fault is detected, a "retry" signal is send to the module to re-initialize its state to the checkpoint state stored in the memory.

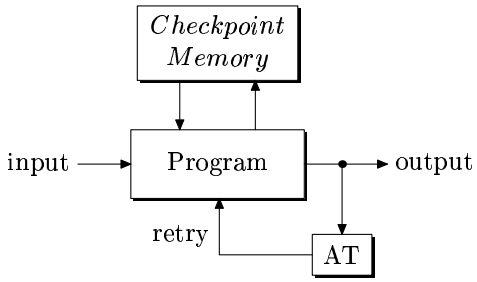

Figure 7.1. Checkpoint and restart recovery.

There are two types of checkpoints: static and dynamic. A static checkpoint takes a single snapshot of the system state at the beginning of the program execution and stores it in the memory. Fault detection checks are placed at the output of the module. If a fault is detected, the system returns to this state and starts the execution from the beginning. Dynamic checkpoints are created dynamically at various points during the execution. If a fault is detected, the system returns to the last checkpoint and continues the execution. Fault detection checks need to be embedded in the code and executed before the checkpoints are created.

A number of factors influence the efficiency of checkpointing, including execution requirements, the interval between checkpoints, fault activation rate and overhead associated with creating fault detection checks, checkpoints, recovery, etc. In a static approach, the expected time to complete the execution grows exponentially with the execution requirements. Therefore, static checkpointing is effective only if the processing requirement is relatively small. In a dynamic approach, it is possible to achieve a linear increase in execution time as the processing requirements grow. There are three strategies for dynamic placing of checkpoints:

1 *Equidistant*, which places checkpoints at deterministic fixed time intervals. The time between checkpoints is chosen depending on the expected fault rate.

2 *Modular*, which places checkpoints at the end of the sub-modules in a module, after the fault detection checks for the sub-module are completed. The execution time depends on the distribution of the sub-modules and expected fault rate.

3 *Random*, placing checkpoints at random.

Overall, restart recovery mechanism has the following advantages:

- It is conceptually simple.
- It is independent of the damage caused by a fault.
- It is applicable to unanticipated faults.
- It is general enough to be used at multiple levels in a system.

A problem with restart recovery is that *non-recoverable actions* exist in some systems. These actions are usually associated with external events that cannot be compensated by simply reloading the state and restarting the system. Examples of non-recoverable actions are firing a missile or soldering a pair of wires. The recovery from such actions need to include special treatment, for example by compensating for their consequences (e.g. undoing a solder), or

delaying their output until after additional confirmation checks are completed (e.g. do a friend-or-foe confirmation before firing).

2.3.3 Process pairs

Process pair technique runs two identical versions of the software on separate processors (Figure 7.2). First the primary processor, Processor 1, is active. It executes the program and sends the checkpoint information to the secondary processor, Processor 2. If a fault is detected, the primary processor is switched off. The secondary processor loads the last checkpoint as its starting state and continues the execution. The Processor 1 executes diagnostic checks off-line. If the fault is non-recoverable, the replacement is performed. After returning to service, the repaired processor becomes secondary processor.

The main advantage of process pair technique is that the delivery of service continues uninterrupted after the occurrence of the fault. It is therefore suitable for applications requiring high availability.

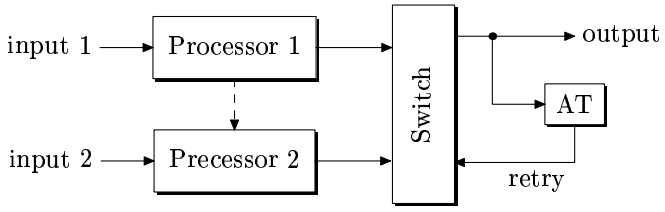

Figure 7.2. Process pairs.

2.3.4 Data diversity

Data diversity is a technique aiming to improve the efficiency of checkpoint and restart by using different inputs re-expressions for each retry. Its is based on the observation that software faults are usually input sequence dependent. Therefore, if inputs are re-expressed in a diverse way, it is unlikely that different re-expressions activate the same fault.

There are three basic techniques for data diversity:

1 Input data re-expression, where only the input is changed.

2 Input data re-expression with post-execution adjustment, where the output result also needs to be adjusted in accordance with a given set of rules. For example, if the inputs were re-expressed by encoding them in some code, then the output result is decoded following the decoding rules of the code.

3 Input data re-expression via decomposition and re-combination, where the input is decomposed into smaller parts and then re-combined after execution to obtain the output result.

Data diversity can also be used in combination with the multi-version fault-tolerance techniques, presented in the next section.

3. Multi-version techniques

Multi-version techniques use two or more versions of the same software module, which satisfy the design diversity requirements. For example, different teams, different coding languages or different algorithms can be used to maximize the probability that all the versions do not have common faults.

3.1 Recovery blocks

The recovery blocks technique combines checkpoint and restart approach with standby sparing redundancy scheme. The basic configuration is shown in Figure 7.3. Versions 1 to *n* represent different implementations of the same program. Only one of the versions provides the system's output. If an error is detected by the acceptance test, a retry signal is sent to the switch. The system is rolled back to the state stored in the checkpoint memory and the switch then switches the execution to another version of the module. Checkpoints are created before a version executes. Various checks are used for acceptance testing of the active version of the module. The check should be kept simple in order to maintain execution speed. Checks can either be placed at the output of a module, or embedded in the code to increase the effectiveness of fault detection.

Similarly to cold and hot versions of hardware standby sparing technique, different versions can be executed either serially, or concurrently, depending on available processing capability and performance requirements. Serial execution may require the use of checkpoints to reload the state before the next version is executed. The cost in time of trying multiple versions serially may be too expensive, especially for a real-time system. However, a concurrent system requires the expense of *n* redundant hardware modules, a communications network to connect them and the use of input and state consistency algorithms.

If all *n* versions are tried and failed, the module invokes the exception handler to communicate to the rest of the system a failure to complete its function.

As all multi-version techniques, recovery blocks technique is heavily dependent on design diversity. The recovery blocks method increases the pressure on the specification to be detailed enough to create different multiple alternatives that are functionally the same. This issue is further discussed in Section 3.4. In addition, acceptance tests suffer from lack of guidelines for their develop-

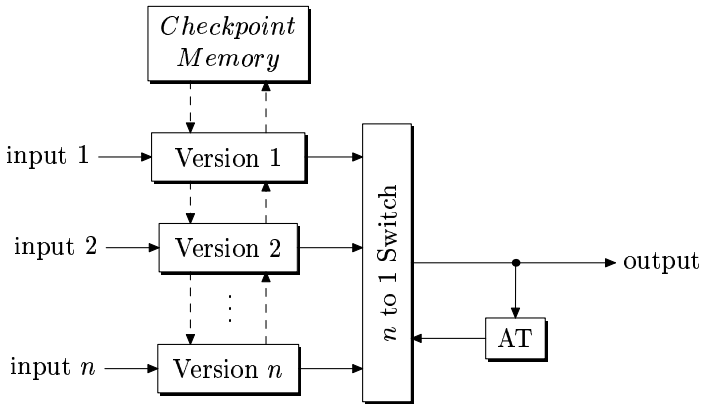

Figure 7.3. Recovery blocks.

ment. They are highly application dependent, they are difficult to create and they cannot test for a specific correct answer, but only for "acceptable" values.

3.2 *N*-version programming

The N-version programming techniques resembles the *N*-modular hardware redundancy. The block diagram is shown in Figure 7.4. It consists of *n* different software implementations of a module, executed concurrently. Each version accomplishes the same task, but in a different way. The selection algorithm decides which of the answers is correct and returns this answer as a result of the module's execution. The selection algorithm is usually implemented as a generic voter. This is an advantage over recovery block fault detection mechanism, requiring application dependent acceptance tests.

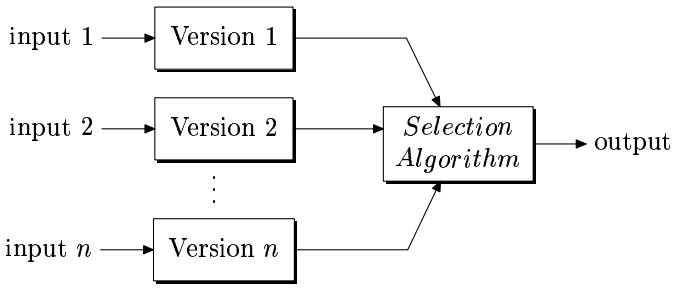

Figure 7.4. *N*-version programming.

Many different types of voters have been developed, including formalized majority voter, generalized median voter, formalized plurality voter and weighted averaging technique. The voters have the capability to perform inexact voting by using the concept of *metric space* (X, d). The set X is the output space of the software and d is a metric function that associates any two elements in X with a real-valued number (see Section 2.5 for the definition of metric). The inexact values are declared equal if their metric distance is less than some pre-defined threshold ε. In the *formalized majority voter*, the outputs are compared and, if more than half of the values agree, the voter output is selected as one of the values in the agreement group. The *generalized median voter* selects the median of the values as the correct result. The median is computed by successively eliminating pair of values that are farther apart until only one value remains. The *formalized plurality voter* partitions the set of outputs based on metric equality and selects the output from the largest partition group. The *weighted averaging technique* combines the outputs in a weighted average to produce the result. The weight can be selected in advance based on the characteristics of the individual versions. If all the weights are equal, this technique reduces to the mean selection technique. The weight can be also selected dynamically based on pair-wise distances of the version outputs or the success history of the versions measured by some performance metric.

The selection algorithms are normally developed taking into account the consequences of erroneous output for dependability attributes like reliability, availability and safety. For applications where reliability is important, the selection algorithm should be designed so that the selected result is correct with a very high probability. If availability is an issue, the selection algorithm is expected to produce an output even if it is incorrect. Such an approach would be acceptable as long as the program execution in not subsequently dependent on previously generated (possibly erroneous) results. For applications where safety is the main concern, the selection algorithm is required to correctly distinguish the erroneous version and mask its results. In cases when the algorithm cannot select the correct result with a high confidence, it should report to the system an error condition or initiate an acceptable safe output sequence.

N-version programming technique can tolerate the design faults present in the software if the design diversity concept is implemented properly. Each version of the module should be implemented in an as diverse as possible manner, including different tool sets, different programming languages, and possibly different environments. The various development groups must have as little interaction related to the programming between them as possible. The specification of the system is required to be detailed enough so that the various versions are completely compatible. On the other hand, the specification should be flexible to give the programmer a possibility to create diverse designs.

3.3 *N* self-checking programming

N self-checking programming combines recovery blocks concept with *N* version programming. The checking is performed either by using acceptance tests, or by using comparison. Examples of applications of *N* self-checking programming are Lucent ESS-5 phone switch and the Airbus A-340 airplane.

N self-checking programming using acceptance tests is shown in Figure 7.5. Different versions of the program module and the acceptance tests AT are developed independently from common requirements. The individual checks for each of the version are either embedded in the code, or placed at the output. The use of separate acceptance tests for each version is the main difference of this technique from recovery blocks approach. The execution of each version can be done either serially, or concurrently. In both cases, the output is taken from the highest-ranking version which passes its acceptance test.

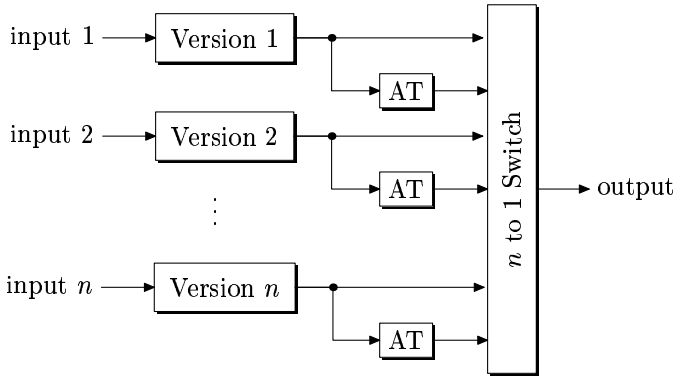

Figure 7.5. *N* self-checking programming using acceptance tests.

N self-checking programming using comparison is shown in Figure 7.6. The scheme resembles triplex-duplex hardware redundancy. An advantage over *N* self-checking programming using acceptance tests is that an application independent decision algorithm (comparison) is used for fault detection.

3.4 Design diversity

The most critical issue in multi-version software fault tolerance techniques is assuring independence between the different versions of software through design diversity. Design diversity aims to protect the software from containing common design faults. Software systems are vulnerable to common design faults if they are developed by the same design team, by applying the same design rules and using the same software tools.

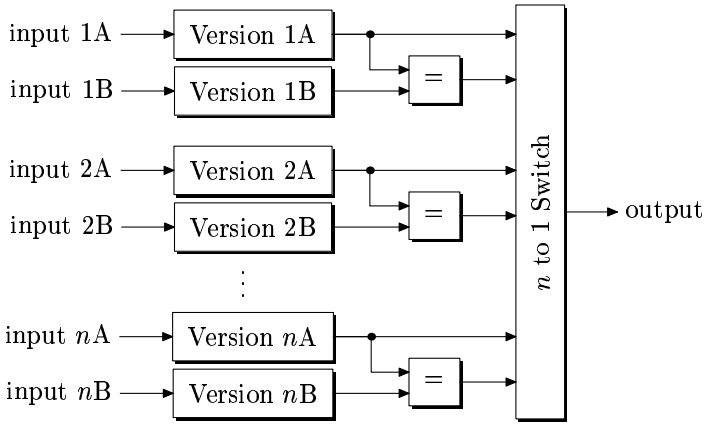

Figure 7.6. *N* self-checking programming using comparison.

Presently, the implementation of design diversity remains a controversial subject. The increase in complexity caused by redundant multiple versions can be quite severe and may result in a less dependent system, unless appropriate measures are taken. Decision to be made when developing a multi-version software system include:

- which modules are to be made redundant (usually less reliable modules are chosen);
- the level of redundancy (procedure, process, whole system);
- the required number of redundant versions;
- the required diversity (diverse specification, algorithm, code, programming language, testing technique, etc.);
- rules of isolation between the development teams, to prevent the flow of information that could result in common design error.

The cost of development of a multi-version software also needs to be taken into account. A direct replication of the full development effort would have a total cost prohibitive for most applications. The cost can be reduced by allocating redundancy to dependability critical parts of the system only. In situations where demonstrating dependability to an official regulatory authority tends to be more costly than the actual development effort, design diversity can be used to make a more dependable system with a smaller safety assessment effort. When the cost of alternative dependability improvement techniques is high because of the need for specialized staff and tools, the use of design diversity can result in cost savings.

4. Software Testing

Software testing is the process of executing a program with the intent of finding errors [Beizer, 1990]. Testing is a major consideration in software development. In many organizations, more time is devoted to testing than to any other phase of software development. On complex projects, test developers might be twice or three times as many as code developers on a project team.

There are two types of software testing: functional and structural. *Functional testing* (also called *behavioral testing*, *black-box testing*, *closed-box testing*), compares test program behavior against its specification. *Structural testing* (also called *white-box testing*, *glass-box testing*) checks the internal structure of a program for errors. For example, suppose we test a program which adds two integers. The goal of functional testing is to verify whether the implemented operation is indeed addition instead of e.g. multiplication. Structural testing does not question the functionally of the program, but checks whether the internal structure is consistent. A strength of the structural approach is that the entire software implementation is taken into account during testing, which facilitates error detection even when the software specification is vague or incomplete.

The effectiveness of structural testing is normally expressed in terms of test coverage metrics, which measure the fraction of code exercised by test cases. Common test coverage metrics are statement, branch, and path coverage [Beizer, 1990]. *Statement* coverage requires that the program under test is run with enough test cases, so that all its statements are executed at least once. *Decision* coverage requires that all branches of the program are executed at least once. *Path* coverage requires that each of the possible paths through the program is followed. Path coverage is the most reliable metric, however, it is not applicable to large systems, since the number of paths is exponential to the number of branches.

This section describes a technique for structural testing which finds a part of program's flowgraph, called *kernel*, with the property that any set of tests which executes all vertices (edges) of the kernel executes all vertices (edges) of the flowgraph [Dubrova, 2005].

Related works include Agarval's algorithm [Agrawal, 1994] for computing the *super block dominator graph* which represents all the kernels of the flowgraph, Bertolino and Marre's algorithm [Bertolino and Marre, 1994] for finding path covers in a flowgraph, in which *unconstrained arcs* are analogous to the leaves of the dominator tree; Ball's [Ball, 1993] and Podgurski's [Podgurski, 1991] techniques for computing *control dependence regions* in a flowgraph, which are similar to the super blocks of [Agrawal, 1994]; Agarwal's algorithm [Agrawal, 1999], which addresses the coverage problem at an interprocedural level.

4.1 Statement and Branch Coverage

This section gives a brief overview of statement and branch coverage techniques.

4.1.1 Statement Coverage

Statement coverage (also called *line coverage, segment coverage* [Ntafos, 1988], *C1* [Beizer, 1990]) examines whether each executable statement of a program is followed during a test. An extension of statement coverage is *basic block coverage*, in which each sequence of non-branching statements is treated as one statement unit.

The main advantage of statement coverage is that it can be applied directly to object code and does not require processing source code. The disadvantages are:

- Statement coverage is insensitive to some control structures, logical AND and OR operators, and switch labels.
- Statement coverage only checks whether the loop body was executed or not. It does not report whether loops reach their termination condition. In C, C++, and Java programs, this limitation affects loops that contain break statements.

As an example of the insensitivity of statement coverage to some control structures, consider the following code:

```
x = 0;
if (condition)
    x = x + 1;
y = 10/x;
```

If there is no test case which causes `condition` to evaluate false, the error in this code will not be detected in spite of 100% statement coverage. The error will appear only if `condition` evaluates false for some test case. Since `if`-statements are common in programs, this problem is a serious drawback of statement coverage.

4.1.2 Branch Coverage

Branch coverage (also referred to as *decision coverage, all-edges coverage* [Roper, 1994], *C2* [Beizer, 1990]) requires that each branch of a program is executed at least once during a test. Boolean expressions of `if`- or `while`-statements are checked to be evaluated to both true and false. The entire Boolean expression is treated as one predicate regardless of whether it contains logical AND and OR operators. `Switch` statements, exception handlers, and interrupt

handlers are treated similarly. Decision coverage includes statement coverage since executing every branch leads to executing every statement.

An advantage of branch coverage is its relative simplicity. It allows overcoming many problems of statement coverage. However, it might miss some errors as demonstrated by the following example:

```
if (condition1)
    x = 0;
else
    x = 2;
if (condition2)
    y = 10*x;
else
    y = 10/x;
```

A 100% branch coverage can be achieved by two test cases which cause both `condition1` and `condition2` to evaluate true, and both `condition1` and `condition2` to evaluate false. However, the error which occurs when `condition1` evaluates true and `condition2` evaluates false will not be detected by these two tests.

The error in the example above can be detected by exercising every path through the program. However, since the number of paths is exponential to the number of branches, testing every path is not possible for large systems. For example, if one test case takes 0.1×10^{-5} seconds to execute, then testing all paths of a program containing 30 `if`-statements will take 18 minutes and testing all paths of a program with 60 `if`-statements will take 366 centuries.

Branch coverage differs from *basic path coverage*, which requires each basis path in the program flowgraph to be executed during a test [Watson, 1996]. Basis paths are a minimal subset of paths that can generate all possible paths by linear combination. The number of basic paths is called the *cyclomatic number* of the flowgraph.

4.2 Preliminaries

A *flowgraph* is a directed graph $G = (V, E, entry, exit)$, where V is the set of vertices representing basic blocks of the program, $E \subseteq V \times V$ is the set of edges connecting the vertices, and *entry* and *exit* are two distinguished vertices of V. Every vertex in V is reachable from *entry* vertex, and *exit* is reachable from every vertex in V.

Figure 7.8 shows the flowgraph of the C program in Figure 7.7, where $bl, b2, \ldots, b16$ are blocks whose contents are not relevant for our purposes.

```
algorithm Example
  b1;
  while(b2) {
      for(b3) {
          b4;
          for(b5) {
              if(b6) b7;
              else b8;
          }
          if(b9) break;
      }
      if(b10) {
          while(b11) b12;
      }
      else {
          if(b13) b14;
          else continue;
      }
      b15;
  }
  b16;
end
```

Figure 7.7. Example C program.

A vertex *v pre-dominates* another vertex *u*, if every path from *entry* to *u* contains *v*. A vertex *v post-dominates* another vertex *u*, if every path from *u* to *exit* contains *v*.

By $Pre(v)$ and $Post(v)$ we denote sets of all nodes which pre-dominate and post-dominate *v*, respectively. E.g. in Figure 7.8, $Pre(5) = \{1,2,3,4\}$ and $Post(5) = \{9,10,16\}$.

Many properties are common for pre- and post-dominators. Further in the paper, we use the word *dominator* to refer to cases which apply to both relationships.

Vertex *v* is the *immediate dominator* of *u*, if *v* dominates *u* and every other dominator of *u* dominates *v*. Every vertex $v \in V$ except *entry* (*exit*) has a unique immediate pre-dominator (post-dominator), $idom(v)$ [Lowry and Medlock, 1969]. For example, in Figure 7.8, vertex 4 is the immediate pre-dominator of 5, and vertex 9 is the immediate post-dominator of 5. The edges $(idom(v), v)$ form a directed tree rooted at *entry* for pre-dominators and at *exit* for post-dominators. Figures 7.9 and 7.10 show the pre- and post-dominator trees of the flowgraph in Figure 7.8.

The problem of finding dominators was first considered in late 60's by Lorry and Medlock [Lowry and Medlock, 1969]. They presented an $O(|V|^4)$ algo-

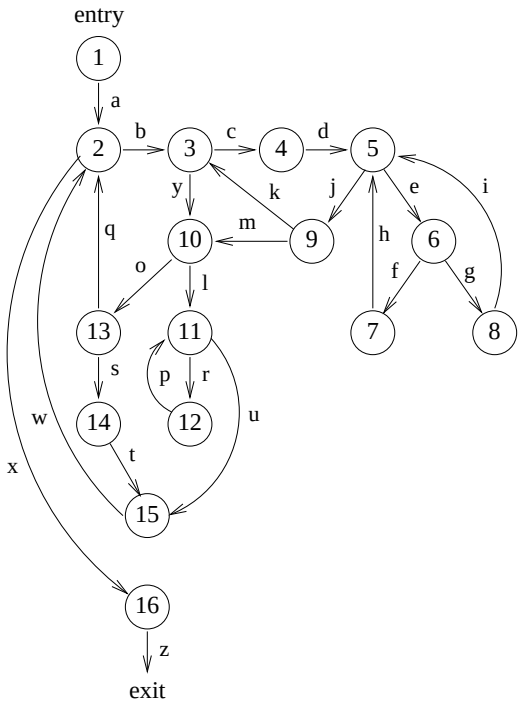

Figure 7.8. Flowgraph of the program in Figure 7.7.

rithm for finding all immediate dominators in a flowgraph. Successive improvements of this algorithm were done by Aho and Ullman [Aho and Ullman, 1972], Purdom and Moore [Purdom and Moore, 1972], Tarjan [Tarjan, 1974], and Lengauer and Tarjan's [Lengauer and Tarjan, 1979]. Lengauer and Tarjan's algorithm [Lengauer and Tarjan, 1979] is a nearly-linear algorithm with the complexity $O(|E|\alpha(|E|,|V|))$, where α is the standard functional inverse of the Ackermann function. Linear algorithms for finding dominators were presented by Harel [Harrel, 1985], Alstrup et al. [Alstrup et al., 1999], and Buchsbaum et al. [Buchsbaum et al., 1998].

4.3 Statement Coverage Using Kernels

This section presents a technique [Dubrova, 2005] for finding a subset of the program's flowgraph vertices, called *kernel*, with the property that any set of tests which executes all vertices of the kernel executes all vertices of the flowgraph. A 100% statement coverage can be achieved by constructing a set of tests for the kernel.

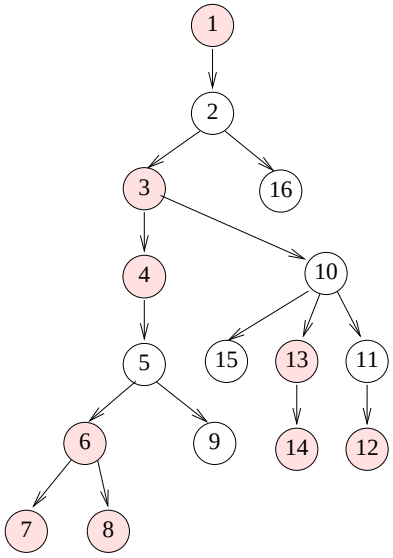

Figure 7.9. Pre-dominator tree of the flowgraph in Figure 7.8; shaded vertices are leaves of the tree in Figure 7.10.

DEFINITION 7.1 *A vertex $v \in V$ of the flowgraph is covered by a test case t if the basic block of the program representing v is reached at least once during the execution of t.*

The following Lemma is the basic property of the presented technique [Agrawal, 1994].

LEMMA 1 *If a test case t covers $u \in V$, then it covers any post-dominator of u as well:*

$$(t \text{ covers } u) \ \wedge \ (v \in Post(u)) \ \Rightarrow \ (t \text{ covers } v).$$

Proof: If v post-dominates u, then every path from u to *exit* contains v. Therefore, if u is reached at least once during the execution of t, then v is reached, too.

DEFINITION 7.2 *A kernel K of a flowgraph G is a subset of vertices of G which satisfies the property that any set of tests which executes all vertices of the kernel executes all vertices of G.*

DEFINITION 7.3 *A minimum kernel is a kernel of the smallest size.*

Let L_{post} (L_{pre}) denote the set of leaf vertices of the post-(pre-)dominator tree of G. The set $L^D_{post} \subset L_{post}$ contains vertices of L_{post} which pre-dominate some

vertex of L_{post}:

$$L^D_{post} = \{v \mid (v \in L_{post}) \wedge (v \in Pre(u) \text{ for some } u \in L_{post})\}.$$

Similarly, the subset $L^D_{pre} \subset L_{pre}$ contains all vertices of L_{pre} which post-dominate some vertex of L_{pre}:

$$L^D_{pre} = \{v \mid (v \in L_{pre}) \wedge (v \in Post(u) \text{ for some } u \in L_{pre})\}.$$

Assume that the program execution terminates normally on all test cases supplied. Then the following statement holds.

THEOREM 7.1 *The set $L_{post} - L^D_{post}$ is a minimum kernel.*

Proof: Lemma 1 shows that, if a vertex of a flowgraph is covered by a test case t, then all its post-dominators are also covered by t. Therefore, in order to cover all vertices of a flowgraph, it is sufficient to cover all leaves L_{post} in its post-dominator tree, i.e. L_{post} is a kernel.

L^D_{post} contains all vertices of L_{post} which pre-dominate some vertex of L_{post}. If v is a pre-dominator of u, and u is covered by t, then v is also covered by t, since every path from *entry* to u contains v as well. Thus, any set of tests which covers $L_{post} - L^D_{post}$, covers L_{post} as well. Since L_{post} is a kernel, $L_{post} - L^D_{post}$ is a kernel, too.

Next, we prove that the set $L_{post} - L^D_{post}$ is a minimum kernel. Suppose that there exists another kernel, K', such that $|K'| < |L_{post} - L^D_{post}|$. If $v \in K'$ and $v \notin L_{post}$, then $v \in Post(u)$ for some $u \in L_{post}$. Since every path from u to *exit* contains v, if u is reached at least once during the execution of some test case, then v is reached, too. Therefore, K' remains a kernel if we replace v by u.

Suppose we replaced all $v \in K'$ such that $v \notin L_{post}$ by $u \in L_{post}$ such that $v \in Post(u)$. Now, $K' \subseteq L_{post}$. If there exists $w \in L_{post} - K'$ such that, for all $u \in K'$, $w \notin Pre(u)$ then there exists at least one path from *entry* to each u which does not contain w. This means that there exists a test set, formed by the set of paths $path(u)$ where $path(u)$ is the path to u which does not contain w, that covers K' but not w. According to Definition 7.2, this implies that K' is not a kernel. Therefore, to guarantee that K' is a kernel, w must be a pre-dominator of some $u \in K'$, for all $w \in L_{post} - K'$. This implies that $|K'| = |L_{post} - L^D_{post}|$.

The next theorem shows that the set $L_{pre} - L^D_{pre}$ is also a minimum kernel.

THEOREM 7.2

$$|L_{post} - L^D_{post}| = |L_{pre} - L^D_{pre}|.$$

The proof is done by showing that the proof of minimality of Theorem 7.1 can be carried out starting from L_{pre}.

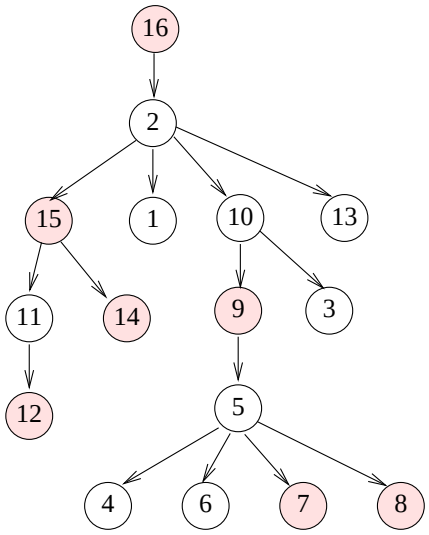

Figure 7.10. Post-dominator tree of the flowgraph in Figure 7.8; shaded vertices are leaves of the tree in Figure 7.9.

4.4 Computing Minimum Kernels

This section presents a linear-time algorithm for computing minimum kernels from [Dubrova, 2005]. The pseudo-code is shown in Figure 7.11.

First, pre- and post-dominator trees of the flowgraph $G = (V, E, entry, exit)$, denoted by T_{pre} and T_{post}, are computed. Then, the numbers of leaves of the trees, L_{pre} and L_{post}, are compared. According to Theorems 7.1 and 7.2, both, $L_{post} - L^D_{post}$ and $L_{pre} - L^D_{pre}$, represent minimum kernels. The procedure FINDLD is applied to the smaller of the sets L_{pre} and L_{post}.

FINDLD checks whether the leaves L_{pre} of the tree T_{pre} are dominated by some vertex of L_{pre} in another tree, T_{post}, or vice versa. In other words, FINDLD computes the set L^D_{pre} (L^D_{post}).

THEOREM 7.3 *The algorithm* KERNEL *computes a minimum kernel of a flowgraph* $G = (V, E, entry, exit)$ *in* $O(|V| + |E|)$ *time.*

Proof: The correctness of the algorithm follows directly from Theorems 7.1 and 7.2. The complexity of the KERNEL is determined by the complexity of computing the T_{pre} and T_{post} trees. A dominator tree can be computed in $O(|V| + |E|)$ time [Alstrup et al., 1999]. Thus, the overall complexity is $O(|V| + |E|)$.

As an example, let us compute a minimum kernel for the flowgraph in Figure 7.8. Its pre- and post-dominator trees are shown in Figures 7.9 and 7.10. T_{pre} has 7 leaves, $L_{pre} = \{7, 8, 9, 12, 14, 15, 16\}$, and T_{post} has 9 leaves, $L_{post} =$

```
algorithm KERNEL(V, E, entry, exit);
  T_pre = PREDOMINATORTREE(V, E, entry);
  L_pre = set of leaves of T_pre;
  T_post = POSTDOMINATORTREE(V, E, exit);
  L_post = set of leaves of T_post;
  if L_pre < L_post then
    K = L_pre − FINDLD(L_pre, T_post);
  else
    K = L_post − FINDLD(L_post, T_pre);
  return K;
end
```

Figure 7.11. Pseudo-code of the algorithm for computing minimum kernels.

$\{1,3,4,6,7,8,12,13,14\}$. So, we check which of the leaves of T_{pre} dominates at least one other leaf of T_{pre} in T_{post}. Leaves L_{pre} are marked as shaded circles in T_{post} in Figure 7.10. We can see that, in T_{post}, vertex 9 dominates 7 and 8, vertex 15 dominates 12 and 14, and vertex 16 dominates all other leaves of L_{pre}. Thus, $L^D_{pre} = \{9,15,16\}$. The minimum kernel $L_{pre} - L^D_{pre}$ consist of four vertices: 7,8,12 and 14.

For a comparison, let us compute the minimum kernel given by $L_{post} - L^D_{post}$. The L_{post} leaves are marked as shaded circles in T_{pre} in Figure 7.9. We can see that, in T_{pre}, vertex 4 dominates 6, 7 and 8, vertex 6 dominates 7 and 8, vertex 13 dominates 14, vertex 3 dominates all leaves of L_{post} except 1, and vertex 1 dominates all leaves of L_{post}. Thus, $L^D_{post} = \{1,3,4,6,13\}$. The minimum kernel $L_{post} - L^D_{post}$ consist of four vertices: 7,8,12 and 14. In this example, the kernels $L_{pre} - L^D_{pre}$ and $L_{post} - L^D_{post}$ are the same, but it is not always the case.

4.5 Decision Coverage Using Kernels

The kernel-based technique described above can be similarly applied to branch coverage by constructing pre- and post-dominator trees for the edges of the flowgraph instead of for its vertices. Figures 7.12 and 7.13 show edge pre- and post-dominator tree of the flowgraph in Figure 7.8.

Similarly to Definition 7.2, a kernel set for edges is defined as a subset of edges of the flowgraph which satisfies the property that any set of tests which executes all edges of the kernel executes all edges of the flowgraph. A 100% branch coverage can be achieved by constructing a set of tests for the kernel. Minimum kernels for Figures 7.12 and 7.13 are: $L_{pre} - L^D_{pre} = \{i,h,k,m,p,q,t,y\}$ and $L_{post} - L^D_{post} = \{f,g,k,r,q,s,m,y\}$.

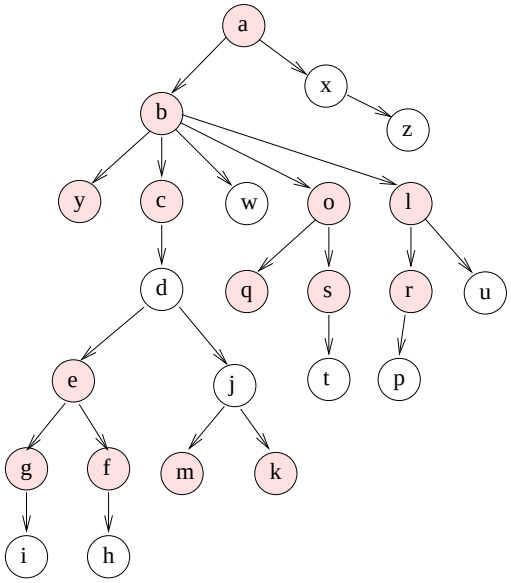

Figure 7.12. Edge pre-dominator tree of the flowgraph in Figure 7.8; shaded vertices are leaves of the tree in Figure 7.13.

5. Problems

7.1. A program consists of 10 independent routines, all of them being used in the normal operation of the program. The probability that a routine is faulty is 0.10 for each of the routines. It is intended to use 3-version programming, with voting to be conducted after the execution of each routine. The effectiveness of the voting in eliminating faults is 0.85 when one of the three routines is faulty and 0 when more than one routine is faulty. What is the probability of a fault-free program:

(a) When only a single version is produced and no routine testing is conducted.

(b) When only a single version of each routine is used, but extensive routine testing is conducted that reduces the fault content to 10% of the original level.

(c) When three-version programming is used.

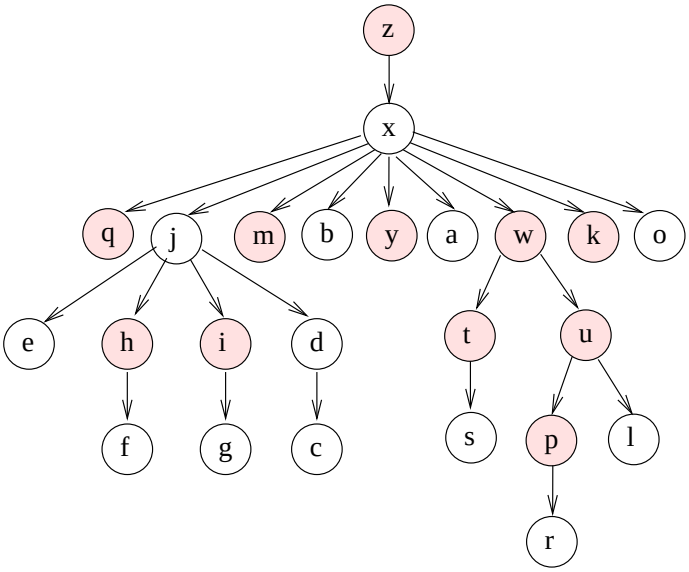

Figure 7.13. Edge post-dominator tree of the flowgraph in Figure 7.8; shaded vertices are leaves of the tree in Figure 7.12.

References

[Agrawal, 1994] Agrawal, H. (1994). Dominators, super blocks, and program coverage. In *Symposium on principles of programming languages*, pages 25–34, Portland, Oregon.

[Agrawal, 1999] Agrawal, H. (1999). Efficient coverage testing using global dominator graphs. In *Workshop on Program Analysis for Software Tools and Engineering*, pages 11–20, Toulouse, France.

[Aho and Ullman, 1972] Aho, A. V. and Ullman, J. D. (1972). *The Theory of Parsing, Translating, and Compiling, Vol. II*. Prentice-Hall, Englewood Cliffs, NJ.

[Alstrup et al., 1999] Alstrup, S., Harel, D., Lauridsen, P. W., and Thorup, M. (1999). Dominators in linear time. *SIAM Journal on Computing*, 28(6):2117–2132.

[Ball, 1993] Ball, T. (1993). What's in a region?: or computing control dependence regions in near-linear time for reducible control flow. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 2:1–16.

[Beizer, 1990] Beizer, B. (1990). *Software Testing Techniques*. Van Nostrand Reinhold, New York.

[Bertolino and Marre, 1994] Bertolino, A. and Marre, M. (1994). Automatic generation of path covers based on the control flow analysis of computer programs. *IEEE Transactions on Software Engineering*, 20:885 – 899.

[Buchsbaum et al., 1998] Buchsbaum, A. L., Kaplan, H., Rogers, A., and Westbrook, J. R. (1998). A new, simpler linear-time dominators algorithm. *ACM Transactions on Programming Languages and Systems*, 20(6):1265–1296.

[Dubrova, 2005] Dubrova, E. (2005). Structural testing based on minimum kernels. In *Proceedings of DATE'2005*, Munich, Germany.

[Harrel, 1985] Harrel, D. (1985). A linear time algorithm for finding dominators in flow graphs and related problems. *Annual Symposium on Theory of Computing*, 17(1):185–194.

[Lengauer and Tarjan, 1979] Lengauer, T. and Tarjan, R. E. (1979). A fast algorithm for finding dominators in a flowgraph. *Transactions of Programming Languages and Systems*, 1(1):121–141.

[Lowry and Medlock, 1969] Lowry, E. S. and Medlock, C. W. (1969). Object code optimization. *Communications of the ACM*, 12(1):13–22.

[Ntafos, 1988] Ntafos, S. (1988). A comparison of some structural testing strategies. *IEEE Transactions on Software Engineering*, 14(6):868–874.

[Podgurski, 1991] Podgurski, A. (1991). Forward control dependence, chain equivalence, and their preservation by reordering transformations. Technical Report CES-91- 18, Computer Engineering & Science Department, Case Western Reserve University, Cleveland, Ohio, USA.

[Purdom and Moore, 1972] Purdom, P. W. and Moore, E. F. (1972). Immediate predominators in a directed graph. *Communications of the ACM*, 15(8):777–778.

[Roper, 1994] Roper, M. (1994). *Software Testing*. McGraw-Hill Book Company, London.

[Tarjan, 1974] Tarjan, R. E. (1974). Finding dominators in a directed graphs. *Journal of Computing*, 3(1):62–89.

[Watson, 1996] Watson, A. H. (1996). Structured testing: Analysis and extensions. Technical Report TR-528-96, Princeton University.